PRATHYUSHA ENGINEERING COLLEGE

LECTURE NOTES

Course code	**: CS8691**
Name of the course	**: ARTIFICIAL INTELLIGENCE**
Regulation	**: 2017**
Course faculty	**: Dr.KAVITHA.V.R**

PRATHYUSHA ENGINEERING COLLEGE
Department of Computer Science and Engineering
VI Semester (EVEN 2019-20)

CS8691ARTIFICIAL INTELLIGENCE

OBJECTIVES:

•To understand the various characteristics of Intelligent agents
•To learnthe different search strategies in AI
•To learn to represent knowledge in solving AI problems
•To understand the different ways of designing software agents
•To know about the various applications of AI.

UNIT IINTRODUCTION 09

Introduction–Definition -Future of Artificial Intelligence –Characteristics of Intelligent Agents–Typical Intelligent Agents –Problem Solving Approach to Typical AI problems.

UNIT II PROBLEM SOLVING METHODS 9

Problem solving Methods -Search Strategies-Uninformed -Informed -Heuristics -Local Search Algorithms and Optimization Problems -Searching with Partial Observations -Constraint Satisfaction Problems –Constraint Propagation -Backtracking Search -Game Playing -Optimal Decisions in Games –Alpha -Beta Pruning - Stochastic Games

UNIT IIIKNOWLEDGE REPRESENTATION 9

First Order Predicate Logic –Prolog Programming –Unification –Forward Chaining-Backward Chaining –Resolution –Knowledge Representation -Ontological Engineering-Categories and Objects –Events -Mental Events and Mental Objects -Reasoning Systems for Categories -Reasoning with Default Information

UNIT IV SOFTWARE AGENTS 9

Architecture for Intelligent Agents –Agent communication –Negotiation and Bargaining –Argumentation among Agents –Trust and Reputation in Multi-agent systems.

UNIT V APPLICATIONS 9

AI applications –Language Models –Information Retrieval-Information Extraction –Natural Language Processing -Machine Translation –Speech Recognition –Robot –Hardware –Perception –Planning –Moving
TOTAL :45 PERIODS

OUTCOMES:

Upon completion of the course, the students will be able to:
•Use appropriate search algorithms for any AI problem
•Represent a problem using first order and predicate logic
•Provide the apt agent strategy to solve a given problem
•Design software agents to solve a problem
•Design applications for NLP that useArtificial Intelligence.

TEXT BOOKS:

1.S. Russell and P. Norvig, "Artificial Intelligence: A Modern Approach, Prentice Hall, Third Edition, 2009.
2. Bratko, ―Prolog: Programming for Artificial Intelligence, Fourth edition, Addison-Wesley Educational Publishers Inc., 2011.

REFERENCES:
1.M. Tim Jones, ―Artificial Intelligence: A Systems Approach(Computer Science), Jones and Bartlett Publishers, Inc.; First Edition, 2008
2.Nils J. Nilsson, ―The Quest for Artificial Intelligence, Cambridge University Press, 2009.
3.William F. Clocksin and Christopher S. Mellish, Programming in Prolog: Using the ISO Standard, Fifth Edition, Springer, 2003.
4.Gerhard Weiss, ―Multi Agent Systems, Second Edition, MIT Press, 2013.
5.David L. Poole and Alan K. Mackworth, ―Artificial Intelligence: Foundations of Computational Agents, Cambridge University Press, 2010

Introduction–Definition -Future of Artificial Intelligence –Characteristics of Intelligent Agents–Typical Intelligent Agents –Problem Solving Approach to Typical AI problems.

1. INTRODUCTION

Artificial intelligence is already all around us, from web search to video games. AI methods plan the driving directions, filter email spam, and focus the digital cameras on faces. AI lets us guide our phone with our voice and read foreign newspapers in English. Beyond today's applications, AI is at the core of many new technologies that will shape our future. From self-driving cars to household robots, advancements in AI help transform science fiction into real systems

The foundations of Artificial Intelligence

The various disciplines that contributed ideas, viewpoints, and techniques to AI are given below :

Philosophy(428 B.C. – present)

Aristotle (384-322 B.C.) was the first to formulate a precise set of laws governing the rational part of the mind. He developed an informal system of syllogisms for proper reasoning, which allowed one to generate conclusions mechanically, given initial premises.

	Computer	Human Brain
Computational units	1 CPU, 10^8 gates	10^{11} neurons
Storage units	10^{10} bits RAM	10^{11} neurons
	10^{11} bits disk	10^{14} synapses
Cycle time	10^{-9} sec	10^{-3} sec
Bandwidth	10^{10} bits/sec	10^{14} bits/sec
Memory updates/sec	10^9	10^{14}

Table 1.1 A crude comparison of the raw computational resources available to computers (*circa* 2003) and brain. The computer's numbers have increased by at least by a factor of 10 every few years. The brain's numbers have not changed for the last 10, 000 years.

Brains and digital computers perform quite different tasks and have different properties. Table 1.1 shows that there are 10000 times more neurons in the typical human brain than there are gates in the CPU of a typical high-end computer. Moore's Law predicts that the CPU's gate count will equal the brain's neuron count around 2020.

Psycology(1879 – present)

The origin of scientific psychology are traced back to the work of German physiologist Hermann von Helmholtz(1821-1894) and his student Wilhelm Wundt(1832 – 1920)

In 1879, Wundt opened the first laboratory of experimental psychology at the University of Leipzig.

In US, the development of computer modeling led to the creation of the field of **cognitive science**.

The field can be said to have started at the workshop in September 1956 at MIT.

Computer engineering (1940-present)

For artificial intelligence to succeed, we need two things: intelligence and an artifact. The computer has been the artifact of choice.

AI also owes a debt to the software side of computer science, which has supplied the operating systems, programming languages, and tools needed to write modern programs

Control theory and Cybernetics (1948-present)

Ktesibios of Alexandria (c. 250 B.C.) built the first self-controlling machine: a water clock with a regulator that kept the flow of water running through it at a constant, predictable pace.

Modern control theory, especially the branch known as stochastic optimal control, has as its goal the design of systems that maximize an **objective function** over time.

Linguistics (1957-present)
Modem linguistics and AI, then, were "born" at about the same time, and grew up together, intersecting in a hybrid field called **computational linguistics** or **natural language processing.**

1.1.3 The History of Artificial Intelligence

The gestation of artificial intelligence (1943-1955)
There were a number of early examples of work that can be characterized as AI, but it was Alan Turing who first articulated a complete vision of AI in his 1950 article "Computing Machinery and Intelligence." Therein, he introduced the Turing test, machine learning, genetic algorithms, and reinforcement learning.

The birth of artificial intelligence (1956)
McCarthy convinced Minsky, Claude Shannon, and Nathaniel Rochester to help him bring together U.S. researchers interested in automata theory, neural nets, and the study of intelligence. They organized a two-month workshop at Dartmouth in the summer of 1956. Perhaps the longest-lasting thing to come out of the workshop was an agreement to adopt McCarthy's new name for the field: **artificial intelligence.**

Early enthusiasm, great expectations (1952-1969)
The early years of AI were full of successes-in a limited way.
General Problem Solver (**GPS**) was a computer program created in 1957 by Herbert Simon and Allen Newell to build a universal problem solver machine. The order in which the program considered subgoals and possible actions was similar to that in which humans approached the same problems. Thus, GPS was probably the first program to embody the "thinking humanly" approach.

At IBM, Nathaniel Rochester and his colleagues produced some of the first AI programs. Herbert Gelernter (1959) constructed the **Geometry Theorem Prover**, which was able to prove theorems that many students of mathematics would find quite tricky.

Lisp was invented by John McCarthy in 1958 while he was at the Massachusetts Institute of Technology (MIT). In 1963, McCarthy started the AI lab at Stanford.

Tom Evans's ANALOGY program (1968) solved geometric analogy problems that appear in IQ tests, such as the one in Figure 1.1

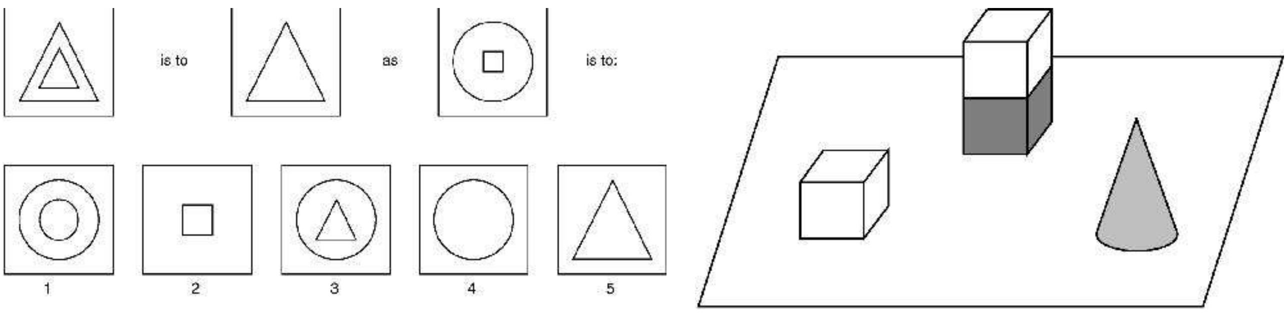


Figure 1.1 The Tom Evan’s ANALOGY program could solve geometric analogy problems as shown.

A dose of reality (1966-1973)
From the beginning, AI researchers were not shy about making predictions of their coming successes. The following statement by Herbert Simon in 1957 is often quoted:
"It is not my aim to surprise or shock you-but the simplest way I can summarize is to say that there are now in the world machines **that think, that learn and that create**. Moreover, their ability to do these things is going to increase rapidly until-in a visible future-the range of problems they can handle will be **coextensive with the range** to which the human mind has been applied.

Knowledge-based systems: The key to power? (1969-1979)
Dendral was an influential pioneer project in artificial intelligence (AI) of the 1960s, and the computer software **expert system** that it produced. Its primary aim was to help organic chemists in identifying unknown organic molecules, by analyzing their mass spectra and using knowledge of chemistry. It was done at Stanford University by Edward Feigenbaum, Bruce Buchanan, Joshua Lederberg, and Carl Djerassi.

AI becomes an industry (1980-present)
In 1981, the Japanese announced the "Fifth Generation" project, a 10-year plan to build intelligent computers running **Prolog**. Overall, the AI industry boomed from a few million dollars in 1980 to billions of dollars in 1988.

The return of neural networks (1986-present)
Psychologists including David Rumelhart and Geoff Hinton continued the study of neural-net models of memory.

AI becomes a science (1987-present)
In recent years, approaches based on **hidden Markov models** (HMMs) have come to dominate the area. **Speech technology and the related field of handwritten character recognition** are already making the transition to widespread industrial and consumer applications.
The **Bayesian network** formalism was invented to allow efficient representation of, and rigorous reasoning with, uncertain knowledge.

The emergence of intelligent agents (1995-present)
One of the most important environments for intelligent agents is the Internet.

2. DEFINITION

What is artificial intelligence?
Artificial Intelligence **is the branch of computer science concerned with making computers behave like humans.**

Intelligent agent **is a system that** perceives **its** environment **and** takes actions **which maximize its chances of success.** John McCarthy**, who coined the term in 1956, defines it as "the science and engineering of making intelligent machines, especially intelligent computer programs."**

The definitions of AI according to some text books are categorized into four approaches and are summarized in the table below:

Systems that think like humans “The exciting new effort to make computers think … machines with minds, in the full and literal sense.”(Haugeland, 1985)	**Systems that think rationally** “The study of mental faculties through the use of computer models.” (Charniak and McDermont, 1985)
Systems that act like humans The art of creating machines that perform functions that require intelligence when performed by people.”(Kurzweil, 1990)	**Systems that act rationally** **“**Computational intelligence is the study of the design of intelligent agents.”(Poole et al., 1998)

<u>Four approaches of AI</u>

(a) Acting humanly : The Turing Test approach

- Test proposed by Alan Turing in 1950
- The computer is asked questions by a human interrogator.

The computer passes the test if a human interrogator, after posing some written questions, cannot tell whether the written responses come from a person or not. Programming a computer to pass, the computer need to possess the following capabilities :

- **Natural language processing** to enable it to communicate successfully in English.
- **Knowledge representation** to store what it knows or hears
- **Automated reasoning** to use the stored information to answer questions and to draw new conclusions.
- **Machine learning** to adapt to new circumstances and to detect and extrapolate patterns

To pass the complete Turing Test, the computer will need

- **Computer vision** to perceive the objects, and
- **Robotics** to manipulate objects and move about.

(b)Thinking humanly: The cognitive modeling approach

We need to get inside actual working of the human mind:

(a) through introspection – trying to capture our own thoughts as they go by;
(b) through psychological experiments

Allen Newell and Herbert Simon, who developed **GPS**, the “**General Problem Solver**” tried to trace the reasoning steps to trace the thought process of human subjects while solving the same problems.
The interdisciplinary field of **cognitive science** brings together computer models from AI and experimental techniques from psychology to try to construct precise and testable theories of the workings of the human mind.

(c) Thinking rationally: The “laws of thought approach”

The Greek philosopher Aristotle was one of the first to attempt to codify “right thinking”, that is irrefutable reasoning processes. His **syllogism** provided patterns for argument structures that always yielded correct conclusions when given correct premises—
for example,

“Ram is student of III year CSE;
All students are good in III year CSE;
Ram is a good student.”

These laws of thought were supposed to govern the operation of the mind; their study initiated a field called **logic.**

(d) Acting rationally: The rational agent approach

An **agent** is something that acts. A **rational agent** is one that acts so as to achieve the best outcome.

Computer agents are not mere programs, but they are expected to have the following attributes also :
(a) operating under autonomous control,
(b) perceiving their environment,
(c) persisting over a prolonged time period,
(d) adapting to change.

3. FUTURE OF ARTIFICIAL INTELLIGENCE

Autonomous planning and scheduling: A hundred million miles from Earth, NASA's Remote Agent program became the first on-board autonomous planning program to control the scheduling of operations for a spacecraft (Jonsson *et* al., 2000). Remote Agent generated plans from high-level goals specified from the ground, and it monitored the operation of the spacecraft as the plans were executed-detecting, diagnosing, and recovering from problems as they occurred.

Game playing: IBM's Deep Blue became the first computer program to defeat the world champion in a chess match when it bested Garry Kasparov by a score of 3.5 to 2.5 in an exhibition match (Goodman and Keene, 1997).

Autonomous control: The ALVINN computer vision system was trained to steer a car to keep it following a lane. It was placed in CMU's NAVLAB computer-controlled minivan and used to navigate across the United States-for 2850 miles it was in control of steering the vehicle 98% of the time.

Diagnosis: Medical diagnosis programs based on probabilistic analysis have been able to perform at the level of an expert physician in several areas of medicine.

Logistics Planning: During the Persian Gulf crisis of 1991, U.S. forces deployed a Dynamic Analysis and Replanning Tool, DART (Cross and Walker, 1994), to do automated logistics planning and scheduling for transportation. This involved up to 50, 000 vehicles, cargo, and people at a time, and had to account for starting points, destinations, routes, and conflict resolution among all parameters. The AI planning techniques allowed a plan to be generated in hours that would have taken weeks with older methods. The Defense Advanced Research Project Agency (DARPA) stated that this single application more than paid back DARPA's 30-year investment in AI.

Robotics: Many surgeons now use robot assistants in microsurgery. HipNav (DiGioia*et* al., 1996) is a system that uses computer vision techniques to create a three-dimensional model of a patient's internal anatomy and then uses robotic control to guide the insertion of a hip replacement prosthesis.

Language understanding and problem solving: PROVERB (Littman et al., 1999) is a computer program that solves crossword puzzles better than most humans, using constraints on possible word fillers, a large database of past puzzles, and a variety of information sources including dictionaries and online databases such as a list of movies and the actors that appear in them.

4. CHARACTERISTICS OF INTELLIGENT AGENTS

The characteristics of intelligent agents

Internal characteristics are

- **Learning/reasoning-** an agent has the ability to learn from previous experience and to successively adapt its own behavior to the environment.
- **Reactivity-** an agent must be capable of reacting appropriately to influences or information from its environment.

- **Autonomy-** an agent must have both control over its actions and internal states. The degree of the agent's autonomy can be specified. There may need intervention from the user only for important decisions.
- **Goal-oriented-** an agent has well-defined goals and gradually influence its environment and so achieve its own goals.

External characteristics are

- **Communication-** an agent often requires an interaction with its environment to fulfill its tasks, such as human, other agents, and arbitrary information sources.
- **Cooperation-** cooperation of several agents permits faster and better solutions for complex tasks that exceed the capabilities of a single agent.
- **Mobility-** an agent may navigate within electronic communication networks.
- **Character-** like human, an agent may demonstrate an external behavior with many human characters as possible.

Intelligent agents have four main characteristics:

"An agent is a computer software system whose characteristics are situatedness, autonomy, adapitvity and sociability."

- **Situatedness**

When an Agent receives some form of sensory input from its environment, it then performs some actions that change its environment in some way. Examples of environments: the physical world and the Internet.

- **Autonomy**

This agent characteristic means that an agent is able to act without direct intervention from humans or other agents. This type of agent has almost complete control over it own actions and internal state.

- **Adapitvity**

This agent characteristic means that it is capable of reacting flexibly to changes within its environment. It is able to accept goal directed initiatives when appropriate and is also capable of learning from it's own experiences, environment and interaction with others.

- **Sociability**

This type of characteristic means that the agent is capable of interacting in a peer-to-peer manner with other agents or humans.

5. TYPICAL INTELLIGENT AGENTS

Agents and environments

An **agent** is anything that can be viewed as perceiving its **environment** through **sensors** and acting upon that environment through **actuators.** This simple idea is illustrated in Figure 1.2.

- A human agent has eyes, ears, and other organs for sensors and hands, legs, mouth, and other body parts for actuators.
- A robotic agent might have cameras and infrared range finders for sensors and various motors for actuators.
- A software agent receives keystrokes, file contents, and network packets as sensory inputs and acts on the environment by displaying on the screen, writing files, and sending network packets.

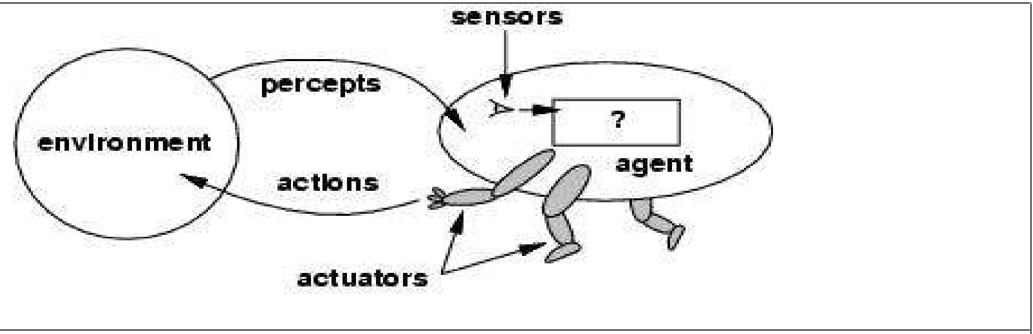


Figure 1.2 Agents interact with environments through sensors and actuators.

Percept

Percept to refer to the agent's perceptual inputs at any given instant.

Percept Sequence

An agent's**percept sequence** is the complete history of everything the agent has ever perceived.

Agent function

Mathematically agent's behavior is described by the **agent function**that maps any given percept sequence to an action.

$$f : \mathcal{P}^* \rightarrow \mathcal{A}$$

Agent program

Internally, The agent function for an artificial agent will be implemented by an **agent program.** It is important to keep these two ideas distinct. The agent function is an abstract mathematical description; the agent program is a concrete implementation, running on the agent architecture.

To illustrate these ideas, we will use a very simple example-the vacuum-cleaner world shown in Figure 1.3. This particular world has just two locations: squares A and B. The vacuum agent perceives which square it is in and whether there is dirt in the square. It can choose to move left, move right, suck up the dirt, or do nothing. One very simple agent function is the following: if the current square is dirty, then suck, otherwise move to the other square. A partial tabulation of this agent function is shown in Figure 1.4.

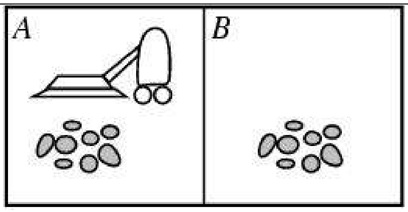


Figure 1.3 A vacuum-cleaner world with just two locations.

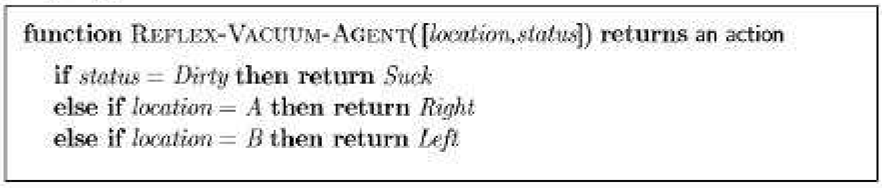
Agent function

Percept Sequence	**Action**
[A, Clean]	Right
[A, Dirty]	Suck
[B, Clean]	Left
[B, Dirty]	Suck

[A, Clean], [A, Clean]	Right
[A, Clean], [A, Dirty]	Suck
…	

Figure 1.4 Partial tabulation of a simple agent function for the vacuum-cleaner world shown in Figure 1.3.

agent program



```
function REFLEX-VACUUM-AGENT([location,status]) returns an action
    if status = Dirty then return Suck
    else if location = A then return Right
    else if location = B then return Left
```

Rational Agent
A **rational agent** is one that does the right thing.Every entry in the table for the agent function is filled out correctly. The right action is the one that will cause the agent to be most successful.
Performance measures
A performance measure embodies the **criterion for success** of an agent's behavior. When an agent is plunked down in an environment, it generates a sequence of actions according to the percepts it receives. This sequence of actions causes the environment to go through a sequence of states. If the sequence is desirable, then the agent has performed well.

Rationality
What is rational at any given time depends on four things:

- The performance measure that defines the criterion of success.
- The agent's prior knowledge of the environment.
- The actions that the agent can perform.
- The agent's percept sequence to date.

Definition of a rational agent:
For each possible percept sequence, a rational agent should select an action that is expected to maximize its performance measure, given the evidence provided by the perceptsequence and whatever built-in knowledge the agent has.

Omniscience, learning, and autonomy
An **omniscient agent** knows the *actual* outcome of its actions and can act accordingly; but omniscience is impossible in reality.
Doing actions in order tomodify future percepts-sometimes called **information gathering**-is an important part of rationality.
Our definition requires a rational agent not only to gather information, but also to **learn**
as much as possible from what it perceives.
To the extent that an agent relies on the prior knowledge of its designer rather than on its own percepts, we say that the agent lacks autonomy. A rational agent should be **autonomous**-it should learn what it can to compensate for partial or incorrect prior knowledge.

1.2.3 The nature of environments
Task environments

We must think about **task environments,** which are essentially the "**problems**" to which rational agents are the "**solutions**."

Specifying the task environment

The rationality of the simple vacuum-cleaner agent, needs specification of

- the **performance** measure
- the **environment**
- the agent's **actuators** and **sensors**.

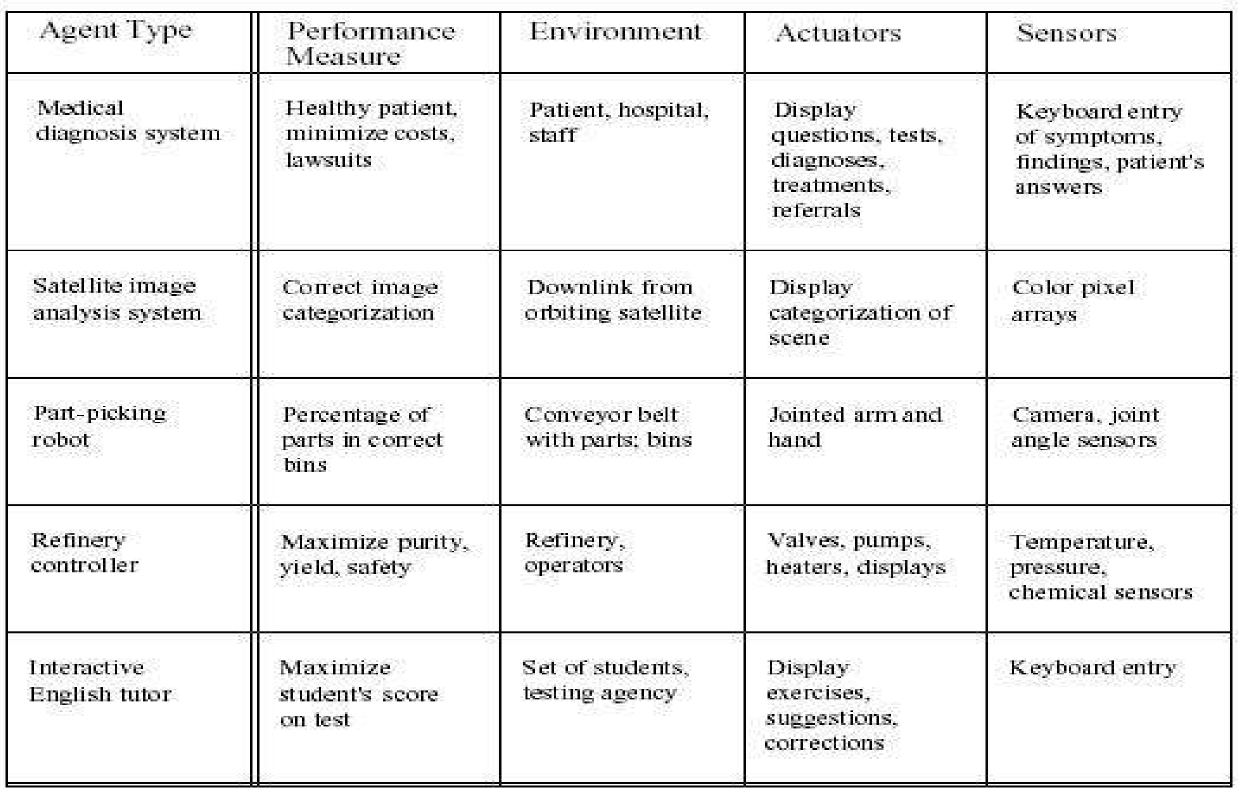
PEAS

All these are grouped together under the heading of the **task environment.** We call this the **PEAS** (Performance, Environment, Actuators, Sensors) description.

In designing an agent, the first step must always be to specify the task environment as fully as possible.

Agent Type	**Performance Measure**	**Environments**	**Actuators**	**Sensors**
Taxi driver	Safe: fast, legal, comfortable trip, maximize profits	Roads, other traffic, pedestrians, customers	Steering, accelerator, brake, Signal, horn, display	Cameras, sonar, Speedometer, GPS, Odometer, engine sensors, keyboards, accelerometer

Figure 1.5 PEAS description of the task environment for an automated taxi.



Agent Type	Performance Measure	Environment	Actuators	Sensors
Medical diagnosis system	Healthy patient, minimize costs, lawsuits	Patient, hospital, staff	Display questions, tests, diagnoses, treatments, referrals	Keyboard entry of symptoms, findings, patient's answers
Satellite image analysis system	Correct image categorization	Downlink from orbiting satellite	Display categorization of scene	Color pixel arrays
Part-picking robot	Percentage of parts in correct bins	Conveyor belt with parts; bins	Jointed arm and hand	Camera, joint angle sensors
Refinery controller	Maximize purity, yield, safety	Refinery, operators	Valves, pumps, heaters, displays	Temperature, pressure, chemical sensors
Interactive English tutor	Maximize student's score on test	Set of students, testing agency	Display exercises, suggestions, corrections	Keyboard entry

Figure 1.6 Examples of agent types and their PEAS descriptions.

<u>**Properties of task environments**</u>

- Fully observable vs. partially observable
- Deterministic vs. stochastic
- Episodic vs. sequential
- Static vs. dynamic
- Discrete vs. continuous

- Single agent vs. multiagent

Fully observable vs. **partially observable.**
If an agent's sensors give it access to the *complete state of the environment at each point in time*, then the task environment is fully observable.
A task environment is effectively fully observable if the sensors detect all aspects that are *relevant* to the choice of action;
An environment might be *partially observable because of noisy and inaccurate sensors* or because parts of the state are simply *missing from the sensor data*.

Deterministic vs. **stochastic.**
If the next state of the environment is *completely determined by the current state* and the action executed by the agent, then we say the environment is deterministic; otherwise, it is **stochastic**.

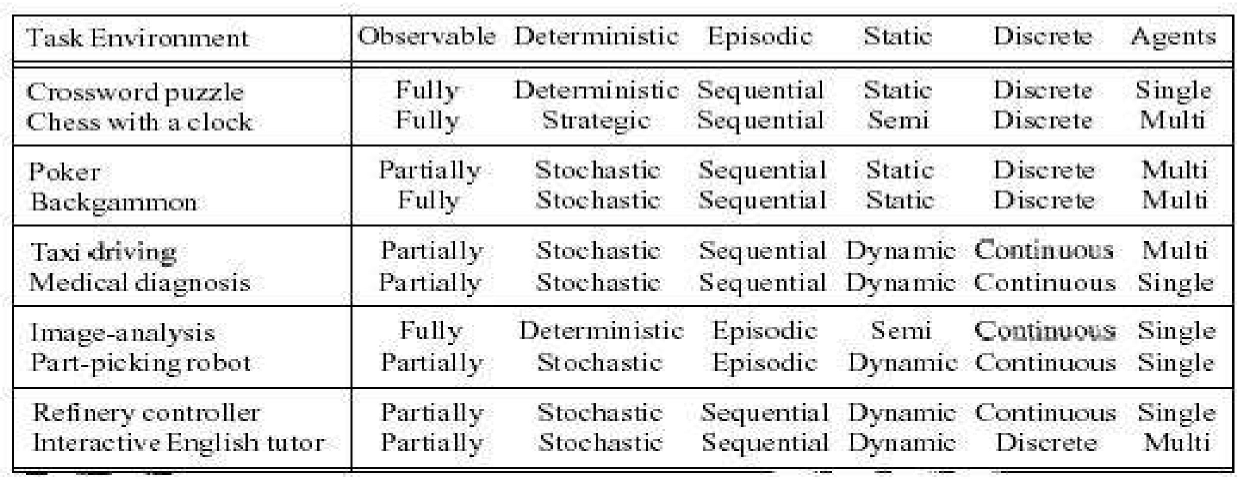
Episodic vs. **sequential**
In an **episodic task environment**, the agent's experience is *divided into atomic episodes*. Each episode consists of the *agent perceiving and then performing a single action*. The next episode does not depend on the actions taken in previous episodes.
For example, an agent that has to spot defective parts on an assembly line bases each decision on the current part, regardless of previous decisions;
In **sequential environments**, on the other hand, the *current decision could affect all future decisions*. Chess and taxi driving are sequential.

Discrete vs. **continuous.**
The discrete/continuous distinction can be applied to the *state* of the environment, to the way *time* is handled, and to the *percepts* and *actions* of the agent.
For example, a discrete-state environment such as a chess game has a finite number of distinct states.Chess also has a discrete set of percepts and actions.

Taxi driving is a *continuous- state and continuous-time problem*: the speed and location of the taxi and of the other vehicles sweep through a range of continuous values and do so smoothly over time.
Taxi-driving actions are also continuous (steering angles, etc.).

Single agent vs. **multiagent.**
An agent solving a crossword puzzle by itself is clearly in asingle-agent environment
Agent playing chess is in a two-agent environment.

The hardest agent is *partially observable, stochastic, sequential, dynamic, continuous,* and *multiagent.*



Task Environment	Observable	Deterministic	Episodic	Static	Discrete	Agents
Crossword puzzle	Fully	Deterministic	Sequential	Static	Discrete	Single
Chess with a clock	Fully	Strategic	Sequential	Semi	Discrete	Multi
Poker	Partially	Stochastic	Sequential	Static	Discrete	Multi
Backgammon	Fully	Stochastic	Sequential	Static	Discrete	Multi
Taxi driving	Partially	Stochastic	Sequential	Dynamic	Continuous	Multi
Medical diagnosis	Partially	Stochastic	Sequential	Dynamic	Continuous	Single
Image-analysis	Fully	Deterministic	Episodic	Semi	Continuous	Single
Part-picking robot	Partially	Stochastic	Episodic	Dynamic	Continuous	Single
Refinery controller	Partially	Stochastic	Sequential	Dynamic	Continuous	Single
Interactive English tutor	Partially	Stochastic	Sequential	Dynamic	Discrete	Multi

Figure 1.7 Examples of task environments and their characteristics.

<u>**Structure of Agents**</u>

Agent programs

The agent programs all have the same skeleton: they take the current percept as input from the sensors and return an action to the actuator. Notice the difference between the **agent program**, which takes the current percept as input, and the **agent function**, which takes the entire percept history. The agent program takes just the current percept as input because nothing more is available from the environment; if the agent's actions depend on the entire percept sequence, the agent will have to remember the percepts.

```
Function TABLE-DRIVEN_AGENT(percept) returns an action

  static: percepts, a sequence initially empty
          table, a table of actions, indexed by percept sequence

  append percept to the end of percepts
  action← LOOKUP(percepts, table)
  returnaction
```

Figure 1.8 The TABLE-DRIVEN-AGENT program is invoked for each new percept and returns**an** action each time.

Drawbacks:

- **Table lookup** of percept-action pairs defining all possible condition-action rules necessary to interact in an environment
- **Problems**
  - Too big to generate and to store (Chess has about 10^120 states, for example)
  - No knowledge of non-perceptual parts of the current state
  - Not adaptive to changes in the environment; requires entire table to be updated if changes occur
  - Looping: Can't make actions conditional
- Take a long time to build the table
- No autonomy
- Even with learning, need a long time to learn the table entries

AGENT TYPES

- **Table-driven agents**
  - use a percept sequence/action table in memory to find the next action. They are implemented by a (large) **lookup table**.
- **Simple reflex agents**

  - are based on **condition-action rules**, implemented with an appropriate production system. They are stateless devices which do not have memory of past world states.
- **Model based agent**
  - have**internal state**, which is used to keep track of past states of the world.
- **Goal based Agents**
  - are agents that, in addition to state information, have **goal information** that describes desirable situations. Agents of this kind take future events into consideration.
- **Utility-based agents**
  - base their decisions on **classic axiomatic utility theory** in order to act rationally.
- **Learning agents**

SIMPLE REFLEX AGENT

The simplest kind of agent is the **simple reflex agent.** These agents select actions on the basis of the *current* percept, ignoring the rest of the percept history. For example, the vacuum agent whose agent function is tabulated in Figure 1.10 is a simple reflex agent, because its decision is based only on the current location and on whether that contains dirt.

- Select action on the basis of *only the current* percept.

E.g. the vacuum-agent

- Large reduction in possible percept/action situations(next page).
- Implemented through *condition-action rules* If dirty then suck

A Simple Reflex Agent: Schema

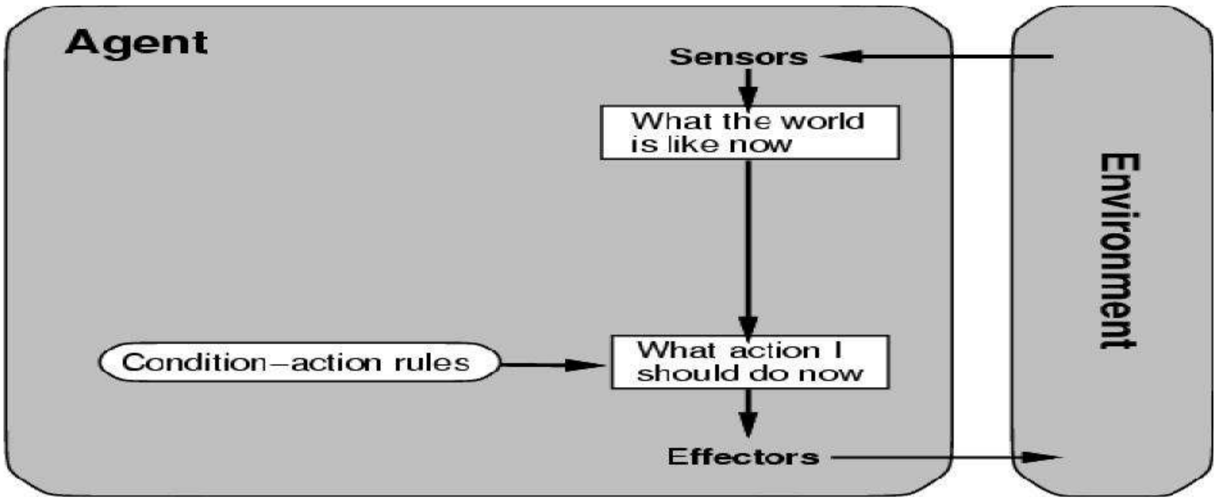


Figure 1.9 Schematic diagram of a simple reflex agent.

```
function SIMPLE-REFLEX-AGENT(percept) returns an action

static: rules, a set of condition-action rules
state← INTERPRET-INPUT(percept)
rule← RULE-MATCH(state, rule)
action← RULE-ACTION[rule]
return action
```

Figure 1.10 A simple reflex agent. It acts according to a rule whose condition matches the current state, as defined by the percept.

```
function REFLEX-VACUUM-AGENT ([location, status]) return an action
  if status == Dirty then return Suck
  else if *location* == *A* then return *Right*
  else if *location* == *B* then return *Left*
```

Figure 1.11 The agent program for a simple reflex agent in the two-state vacuum environment. This program implements the agent function tabulated in the figure 1.4.

❖ **Characteristics**

- Only works if the environment is fully observable.
- Lacking history, easily get stuck in infinite loops
- One solution is to randomize actions

MODEL-BASED REFLEX AGENTS

The most effective way to handle partial observability is for the agent to *keep track of the part of the world it can't see now.* That is, the agent should maintain some sort of **internal state**that depends on the percept history and thereby reflects at least some of the unobserved aspects of the current state.
Updating this internal state information as time goes by requires two kinds of knowledge to be encoded in the agent program.
First, we need some information about *how the world evolves independently* of the agent-for example, that an overtaking car generally will be closer behind than it was a moment ago.
Second, we need some information about how the *agent's own actions affect the world*-for example, that when the agent turns the steering wheel clockwise, the car turns to the right or that after driving for five minutes northbound on the freeway one is usually about five miles north of where one was five minutes ago.
This knowledge about *"how the world working - whether implemented in simple Boolean circuits or in complete scientific theories-is called a **model** of the world.* An agent that uses such a MODEL-BASED model is called a **model-based agent.**

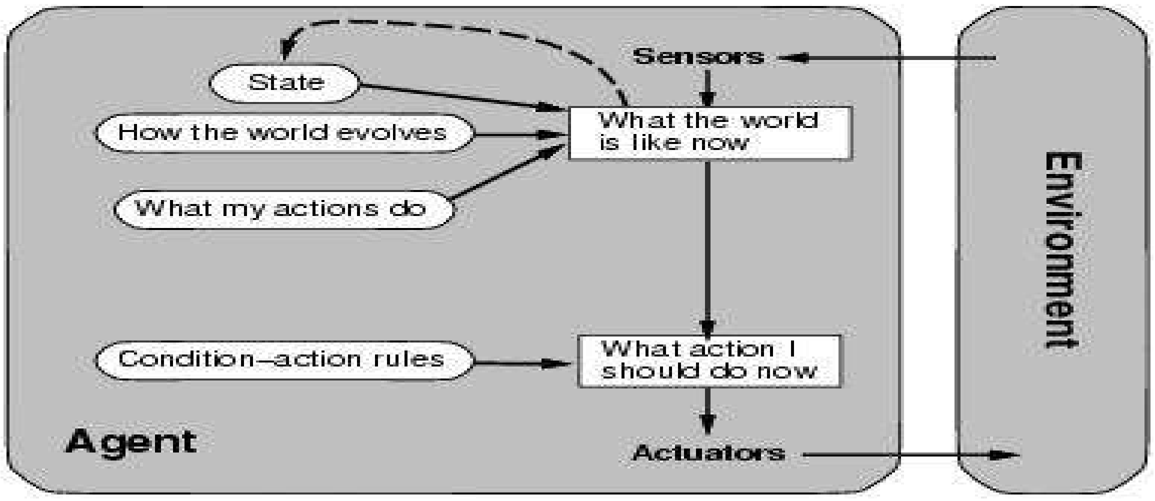


Figure 1.12 A model based reflex agent

```
function REFLEX-AGENT-WITH-STATE(percept) returns an action
static: rules, a set of condition-action rules
state, a description of the current world state
action, the most recent action.
state← UPDATE-STATE(state, action, percept)
rule← RULE-MATCH(state, rule)
action← RULE-ACTION[rule]
return action
```

Figure 1.13 Model based reflex agent. It keeps track of the current state of the world using an internal model. It then chooses an action in the same way as the reflex agent.

GOAL-BASED AGENTS

- Knowing about the current state of the environment is not always enough to decide what to do.
  For example, at a road junction, the taxi can turn left, turn right, or go straight on.
- The correct decision depends on where the taxi is trying to get to.
- In other words, as well as a current state description, the agent needs some sort of **goal** information that describes situations that are desirable-for example, being at the passenger's destination.
- The agent program can combine this with information about the results of possible actions (the same information as was used to update internal state in the reflex agent) in order to choose actions that achieve the goal. Figure 1.14 shows the goal-based agent's structure.

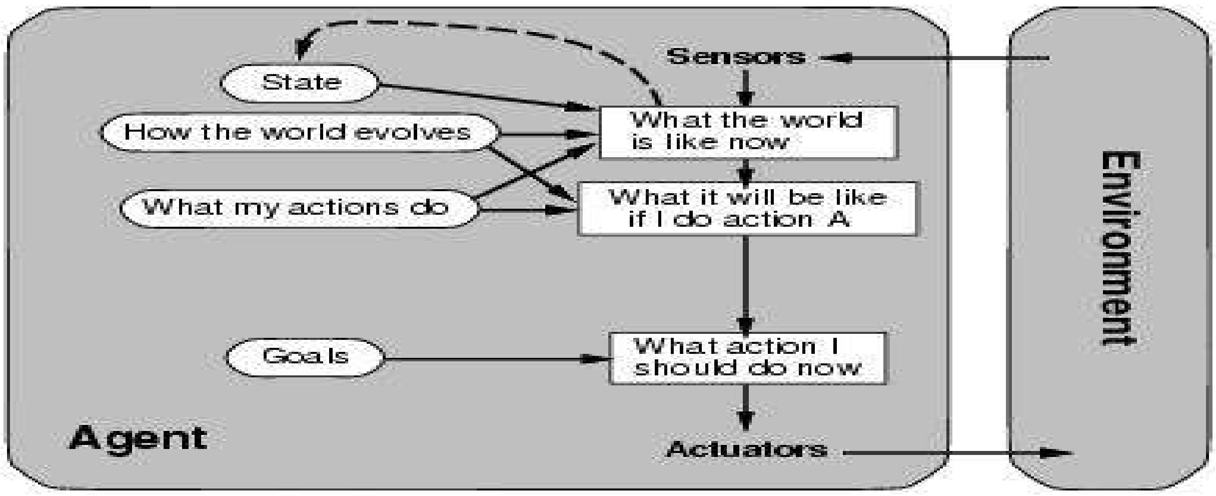

Sensors
State
How the world evolves
What my actions do
What the world is like now
What it will be like if I do action A
Goals
What action I should do now
Agent
Actuators
Environment

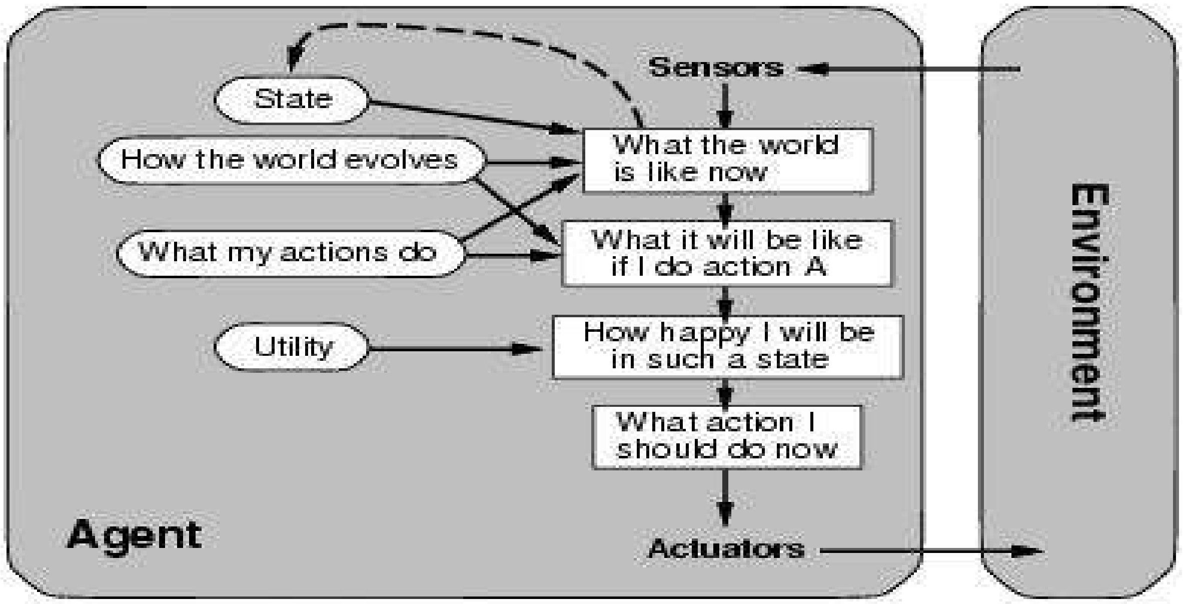
Figure 1.14 A goal based agent

UTILITY-BASED AGENTS

Goals alone are not really enough to generate high-quality behavior in most environments. For example, there are many action sequences that will get the taxi to its destination (thereby achieving the goal) but some are *quicker, safer, more reliable, or cheaper* than others.

Goals just provide a crude binary distinction between "happy" and "unhappy" states, whereas a more general **performance measure** should allow a comparison of different world states according to exactly how happy they would make the agent if they could be achieved.

Because "happy" does not sound very scientific, the customary terminology is to say that if *one world state is preferred to another*, then it has *higher* ***utility*** *for the agent*.



Sensors
State
How the world evolves
What my actions do
Utility
What the world is like now
What it will be like if I do action A
How happy I will be in such a state
What action I should do now
Agent
Actuators
Environment

Figure 1.15 A model-based, utility-based agent. It uses a model of the world, along with utility function that measures its preferences among states of the world. Then it chooses the action that leads to the best expected utility, where expected utility is computed by averaging over all possible outcome states, weighted by the probability of the outcome.

- Certain goals can be reached in different ways.
  - Some are better, have a higher utility.
- Utility function maps a (sequence of) state(s) onto a real number.
- Improves on goals:
  - Selecting between conflicting goals
  - Select appropriately between several goals based on likelihood of success.

LEARNING AGENTS

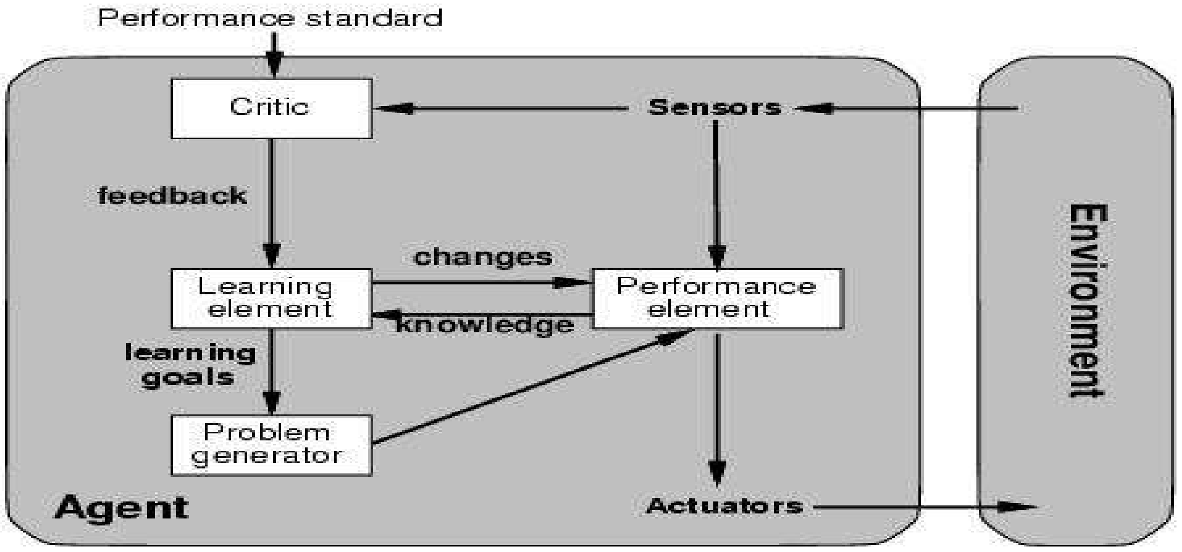


Figure 1.16 A general model of learning agents.

- All agents can improve their performance through **learning.**

A learning agent can be divided into four conceptual components, as shown in Figure 1.16

Performance element: Responsible for selecting external actions. The performance element is what we have previously considered to be the entire agent: it takes in percepts and decides on actions.

Learning element : Responsible for making improvements. It uses feedback from the **critic** on how the agent is doing and determines how the performance element should be modified to do better in the future.

Problem generator: Responsiblefor suggesting actions that will lead to new and **informative experiences**. But if the agent is willing to explore a little, it might discover much better actions for the long run. The problemgenerator's job is to suggest these **exploratory actions**. This is what scientists do when theycarry out experiments.

Use of Intelligent Agents in Businesses

- Operating systems use agents to add email and dial up networking account, do group management, add/remove programs and devices and monitor licences.
- Spreadsheet agents offer suggestions for improvement and can also tutor novice users.
- Software development agents assist in routine activities such as data filtering.
- Search engines – improve your information retrieval on the Internet
- Web mastering Agents – these agents make it easy to manage a web site
- Web Agents – These agents improve the users browsing experience.
- Monitoring Agents – These agents monitor web sites or specific themes you are interested in.
- Shopbots – These agents allow you to compare prices on Internet.
- Virtual Assistants – these include virtual pets and desktop assistants.

6. TYPICAL AI PROBLEMS

PROBLEM FORMULATION

Goal formulation, based on the current situation and the agent's performance measure, is the first step in problem solving. The agent's task is to find out which sequence of actions will get to a goal state.

Problem formulation is the process of deciding what actions and states to consider given a goal.
A state represents a status of the solution at a given step of the problem solving procedure. The solution of a problem, thus, is a collection of the problem states. The problem solving procedure applies an operator to a state to get the next state. Then it applies another operator to the resulting state to derive a new state.
A **solution** to the problem is a path from the initial state to a goal state. An **optimal solution** has the lowest path cost among all solutions.

Formal description of a problem

- Define a **state spac**e that contains all possible configurations of the relevant objects, without enumerating all the states in it. A state space represents a problem in terms of states and operators that change states
- Define some of these states as possible **initial states**;
- Specify one or more as acceptable solutions, these are **goal states**;
  Specify **a set of rules** as the possible actions allowed. This involves thinking about the generality of the rules, the assumptions made in the informal presentation and how much work can be anticipated by inclusion in the rules.

Examples of Problem Definitions

<u>Problem definition for "Play Chess"</u>

The starting position can be specified as an 8x8 array. Each position contains the symbol representing the presence of a particular piece.
Goal position can be defined as any board position in which the opponent does not have a legal move and his/her king is under attack.
The legal moves provide the way to reach the goal state from the initial state and are stated as rules.
These rules can be described as set of rules having two parts. Left side contains the pattern to be matched against the current position. Right side contains the change to be made to the board to represent the move.

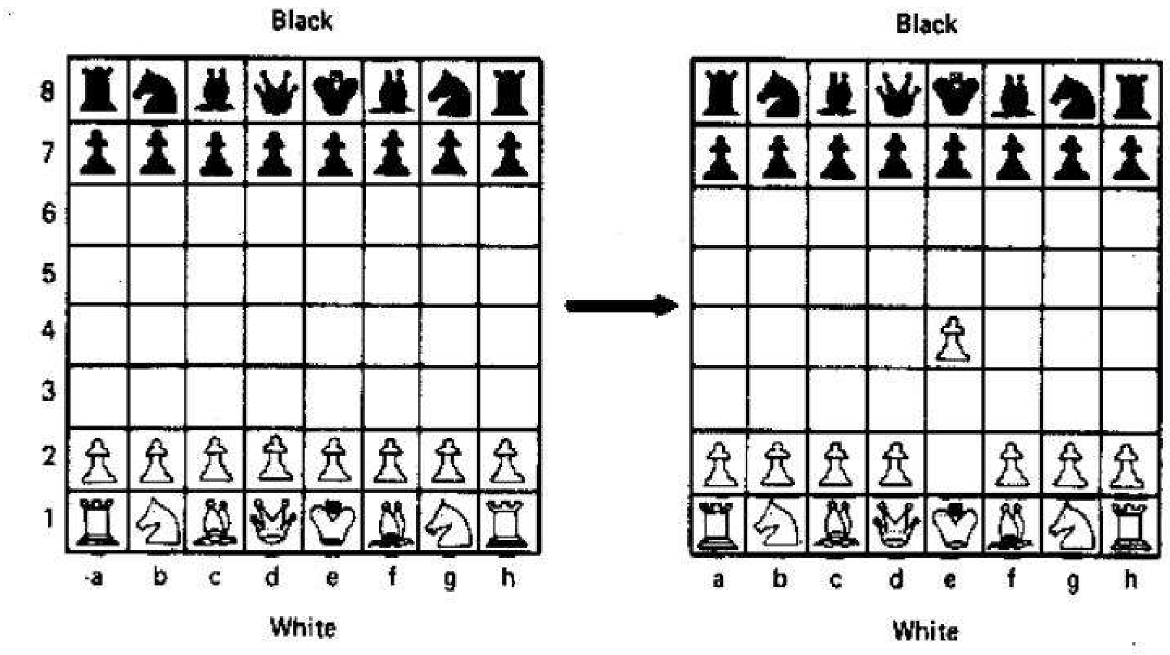


Fig 1.1 One Legal chess board move

Examples :8–Puzzle

◊ State space : configuration of **8 - tiles** on the board
◊ Initial state :any configuration
◊ Goal state :tiles in a specific order
◊ Action :"blank moves"
◊ Condition: the move is within the board
◊ Transformation: blank moves Left, Right, Up, Dn
◊ Solution :optimal sequence of operators

Example : A game of n - queens puzzle; n = 8

State space : configurations **n = 8** queens on the board with only one queen per row and column
◊ Initial state : configuration without queens on the board
◊ Goal state : configuration with **n = 8** queens such that no queen attacks any other
Operators or actions : place a queen on the board.
◊ Condition: the new queen is not attacked by any other already placed
◊ Transformation: place a new queen in a particular square of the board
◊ Solution :one solution (cost is not considered)

Example :

The water jug problem :

A Water Jug Problem: You are given two jugs, a 4-gallon one and a 3-gallon one, a pump which has unlimited water which you can use to fill the jug, and the ground on which water may be poured. Neither jug has any measuring markings on it. How can you get exactly 2 gallons of water in the 4-gallon jug?

State Representation and Initial State–we will represent a state of the problem as a tuple (x,y) where x represents the amount of water in the 4-gallon jug and y represents the amount of water in the 3-gallon jug. Note $0 \leq x \leq 4$, and $0 \leq y \leq 3$. Our initial state: (0,0)

•State: (x,y) x=0,1,2,3,or 4 y =0,1,2,3
•Start state: (0,0).

•Goal state: (2,n) for any n.

•Attempting to end up in a goal state

State Space Search: WaterJug Problem

1	(x, y) if $x < 4$	$\rightarrow$	$(4, y)$	Fill the 4-gallon jug
2	(x, y) if $y < 3$	$\rightarrow$	$(x, 3)$	Fill the 3-gallon jug
3	(x, y) if $x > 0$	$\rightarrow$	$(x - d, y)$	Pour some water out of the 4-gallon jug
4	(x, y) if $y > 0$	$\rightarrow$	$(x, y - d)$	Pour some water out of the 3-gallon jug
5	(x, y) if $x > 0$	$\rightarrow$	$(0, y)$	Empty the 4-gallon jug on the ground
6	(x, y) if $y > 0$	$\rightarrow$	$(x, 0)$	Empty the 3-gallon jug on the ground
7	(x, y) if $x + y \geq 4$ and $y > 0$	$\rightarrow$	$(4, y - (4 - x))$	Pour water from the 3-gallon jug into the 4-gallon jug until the 4-gallon jug is full
8	(x, y) if $x + y \geq 3$ and $x > 0$	$\rightarrow$	$(x - (3 - y), 3)$	Pour water from the 4-gallon jug into the 3-gallon jug until the 3-gallon jug is full
9	(x, y) if $x + y \leq 4$ and $y > 0$	$\rightarrow$	$(x + y, 0)$	Pour all the water from the 3-gallon jug into the 4-gallon jug
10	(x, y) if $x + y \leq 3$ and $x > 0$	$\rightarrow$	$(0, x + y)$	Pour all the water from the 4-gallon jug into the 3-gallon jug
11	$(0, 2)$	$\rightarrow$	$(2, 0)$	Pour the 2 gallons from the 3-gallon jug into the 4-gallon jug
12	$(2, y)$	$\rightarrow$	$(0, y)$	Empty the 2 gallons in the 4-gallon jug on the ground

The role of the condition in the left side of a rule

⇒restrict the application of the rule

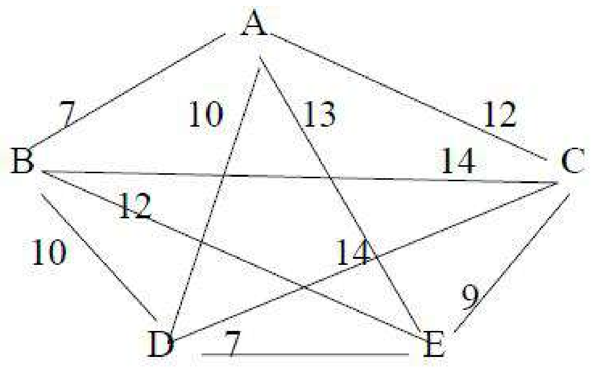
⇒more efficient

One Solution to water jug problem is

Gallons in the 4-Gallon Jug	Gallons in the 3-Gallon Jug	Rule Applied
0	0	
		2
0	3	
		9
3	0	
		2
3	3	
		7
4	2	
		5 or 12
0	2	
		9 or 11
2	0	

Example :Traveling Salesman Problem (with 5 cities):

A salesman is supposed to visit each of 5 cities shown below. There is a road between each pair of cities and the distance is given next to the roads. Start city is A. The problem is to find the shortest route so that the salesman visits each of the cities only once and returns to back to A.

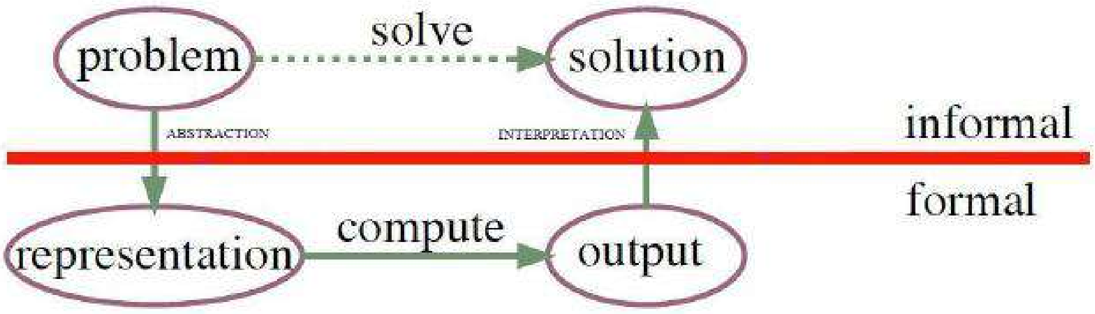


•A simple, motion causing and systematic control structure could, in principle solve this problem.
•Explore the search tree of all possible paths and return the shortest path.
•This will require 4! paths to be examined.
•If number of cities grow, say 25 cities, then the time required to wait a salesman to get the information about the shortest path is of O (24!) which is not a practical situation.
•This phenomenon is called **combinatorial explosion**.
•We can improve the above strategy as follows.

- Begin generating complete paths, keeping track of the shortest path found so far.
- Give up exploring any path as soon as its partial length become greater than the shortest path found so far.

This algorithm is efficient than the first one, still requires exponential time ∝ some number raised to N.

6. PROBLEM SOLVING APPROACH TO TYPICAL AI PROBLEMS.



• Graph Search
– Representation of states and transitions/actionsbetween states → graph
– Explored explicitly or implicitly
• Constraint Solving
– Represent problem by variables and constraints
– Use specific solving algorithms to speedup search
• Local Search and Metaheuristics
– Evaluation function to check if state is "good" or not
– Optimization of the evaluation function

<u>GRAPH SEARCH</u>

- A large variety of problems can berepresented by a graph

- A (directed) graph consists of a set of nodes and a set of directed arcs between nodes.
- The idea is to find a path along these arcs from a start node to a goal node.
- A directed **graph** consists of
- a set *N* of **nodes** and
- a set *A* of ordered pairs of nodes called **arcs**.

A **path** from node *s* to node *g* is a sequence of nodes $\langle n_0, n_1, \ldots, n_k \rangle$
A **tree** is

- a DAG where there is one node with no incoming arcs and every other node has exactly one incoming arc.
- Node with no incoming arcs is called the **root** of the tree
- Nodes with no outgoing arcs are called **leaves**.

To encode problems as graphs,

- One set of nodes is referred to as the **start nodes**
- Another set is called the **goal nodes**.

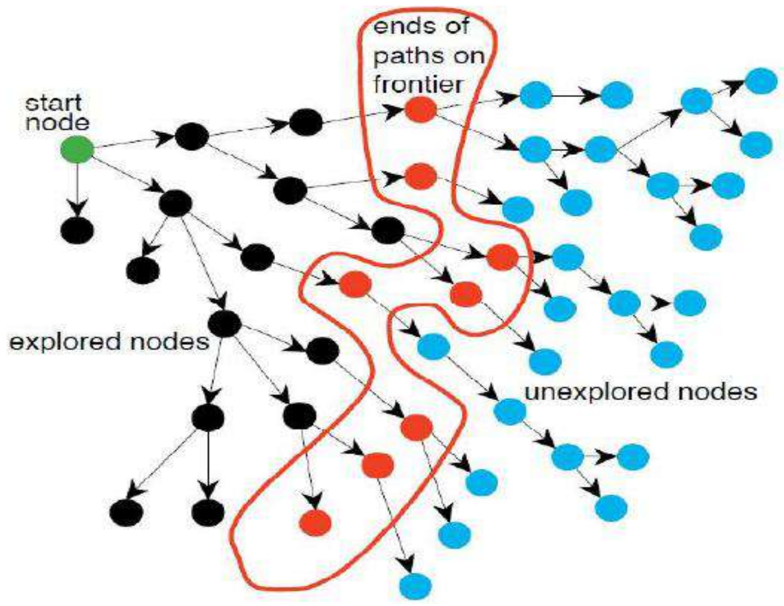
A **solution** is a path from a start node to a goal node.
An **optimal solution** is one of the least-cost solutions; that is, it is a path *p* from a start node to a goal node such that there is no path *p'* from a start node to a goal node where $cost(p')<cost(p)$.
The **forward branching factor** of a node is the number of arcs leaving the node.
The **backward branching factor** of a node is the number of arcs entering the node. These factors provide measures of the complexity of graphs.

• Solutions can be considered as definingspecific nodes
• Solving the problem is reduced to searchingthe graph for those nodes
  – starting from an initial node
  – each transition in the graph corresponds to apossible action
  – ending when reaching a final node (solution)

Defining Search Problems

- A statement of a Search problem has 4 components
  1. A set of states
  2. A set of “operators” which allow one to get from one state to another
  3. A start state S
  4. A set of possible goal states, or ways to test for goal states

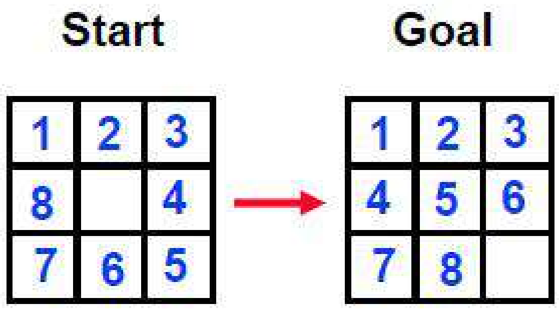
  4a. Cost path
- Search solution consists ofa sequence of operators which transform S into a goal state G



BASIC GRAPH SEARCH ALGORITHM

```
Input: a graph,
       a set of start nodes,
       Boolean procedure $goal(n)$ that tests if $n$ is a goal node.
$frontier := \{\langle s \rangle : s$ is a start node$\}$;
while $frontier$ is not empty:
       select and remove path $\langle n_0, \ldots, n_k \rangle$ from $frontier$;
       if $goal(n_k)$
          return $\langle n_0, \ldots, n_k \rangle$;
       for every neighbor $n$ of $n_k$
          add $\langle n_0, \ldots, n_k, n \rangle$ to $frontier$;
end while
```

<u>**The 8-puzzle**</u>



1	2	3
8		4
7	6	5

1	2	3
4	5	6
7	8	

• can be generalized to15-puzzle, 24-puzzle, etc.

• Any (n2 -1)-puzzle for n ≥ 3

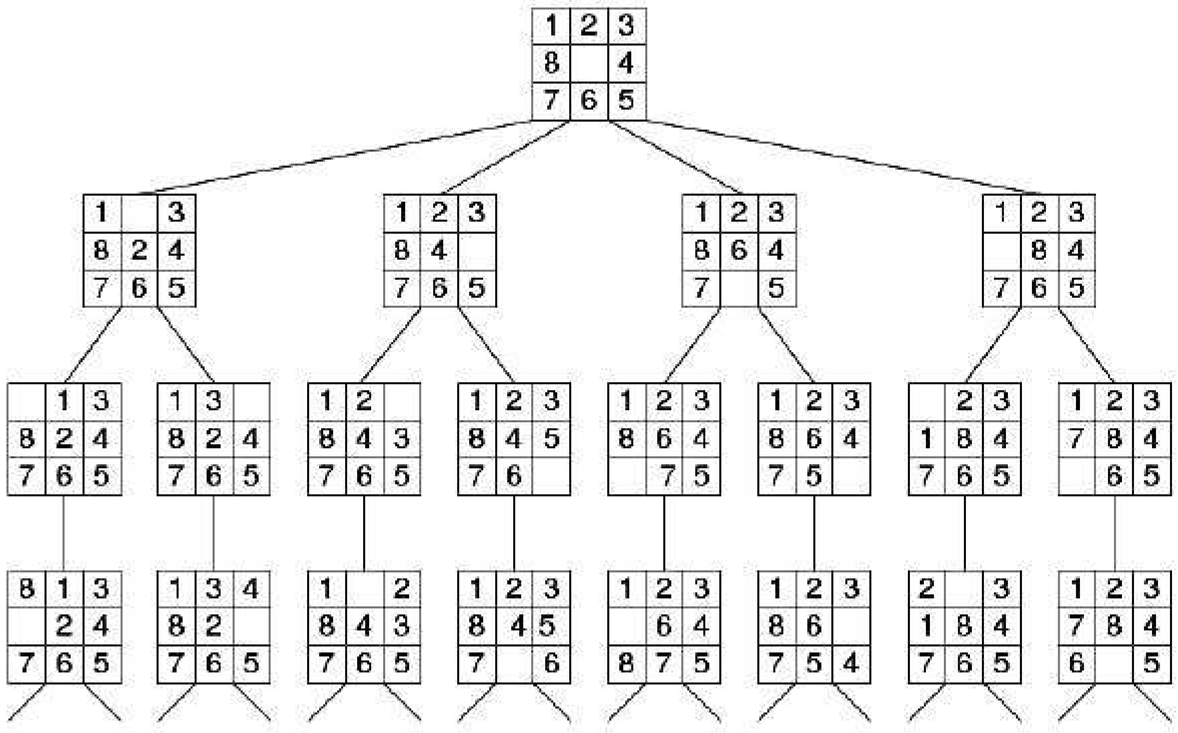
• state = permutation of (0, 1, 2, 3, 4, 5, 6, 7, 8)

Eg.state above is: (1,2,3,8,0,4,7,6,5)

• 9! = 362,880 possible states

• Solution: (1,2,3,4,5,6,7,8,0)
• Actions: possible moves

Search Tree for 8-Puzzle



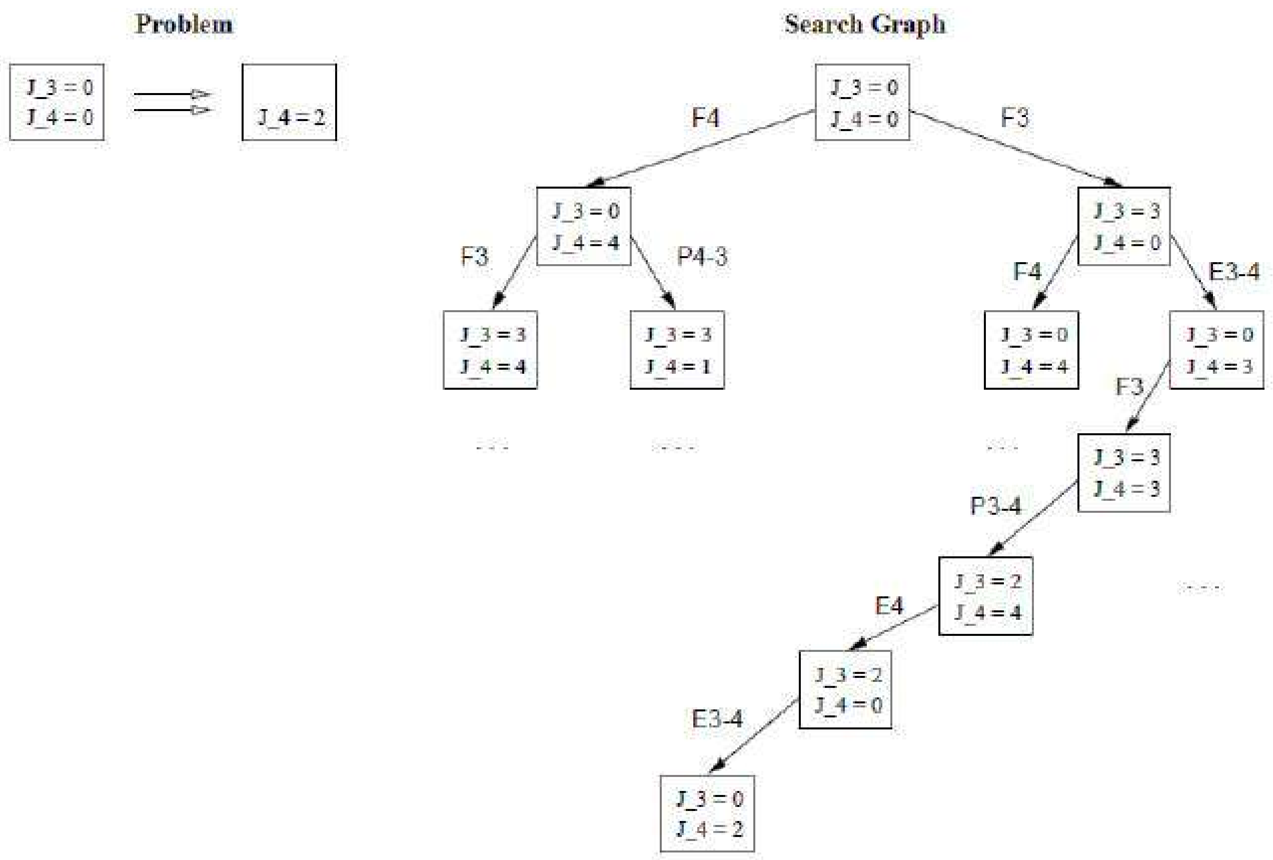
Search:
•Expand out possible nodes
•Maintain a fringe of as yet unexpanded nodes
•Try to expand as few tree nodes as possible

Water Jug Problem

A Water Jug Problem: You are given two jugs, a 4-gallon one and a 3-gallon one, a pump which has unlimited water which you can use to fill the jug, and the ground on which water may be poured. Neither jug has any measuring markings on it. How can you get exactly 2 gallons of water in the 4-gallon jug?

State Representation and Initial State–we will represent a state of the problem as a tuple (x,y) where x represents the amount of water in the 4-gallon jug and y represents the amount of water in the 3-gallon jug. Note $0 \leq x \leq 4$, and $0 \leq y \leq 3$. Our initial state: (0,0)

•State: (x,y)   x=0,1,2,3,or 4   y =0,1,2,3
•Start state: (0,0).
•Goal state: (2,n) for any n.
•Attempting to end up in a goal state



CONSTRAINT MODELING

Search can be made easier in cases where the solution of corresponding to an optimal path, is only required to satisfy local consistency conditions. We call such problems *Constraint Satisfaction (CS) Problems.*

Many problems can be stated as constraints satisfaction problems.

Two-step process:

1. Constraints are discovered and propagated as far as possible.
2. If there is still not a solution, then search begins, adding new constraints.

Two kinds of rules:

1. Rules that define valid constraint propagation.
2. Rules that suggest guesses when necessary.

.

- **Declarative language** : modeling is easy
- **local** specification of the problem
- **global** consistency achieved (or approximated)by constraint solving techniques
- **compositionality** : constraints are combined implicitly through shared logical variables

Basic Objects

Variable : a place holder for values *X*,*Y*, *Z*, *L* ,*U* , *List* 3 21

Function Symbol: mapping of variables to values : Sin(x), Cos(x), x+y, x-y

Relation Symbol:

Relation between variables

Arithmetic relation : +, -, x, /

Symbolic relation : *all_different*

Constraints

Declarative relations between variables

- Constraints used to both model and solve the problem
- specific algorithms for efficient computation Constraints
- Constraints could be numeric or symbolic :

  $X \leq 5$, $X + Y = Z$

  all_different(X1,X2,…,Xn)

  at_most(N,[X1,X2,X3],V)
- multi-directional relations

Crypto-arithmetics as CSP

```
  S E N D
+ M O R E
_________
M O N E Y
```

each letter represents a (different) digit

and the addition should be correct !

... Solution ?

C4 C3 C2 C1

```
     S   E  N  D
+   M   O   R  E
________________
M  O  N   E  Y
```

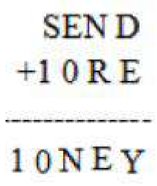
variables :{S,E,N,D,M,O,R,Y,R1, R2, R3, R4}

domains :{0,...,9} for the letters, {0,1} for the carries

constraints : all_different(S,E,N,D,M,O,R,Y} $S \neq 0$ $M \neq 0$

We have to replace each letter by a distinct digit so that the resulting sum is correct.

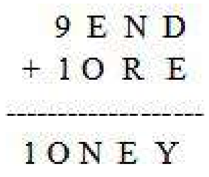
In the above problem, M must be 1. You can visualize that, this is an addition problem. The sum of two four digit numbers cannot be more than 10,000. Also M cannot be zero according to the rules, since it is the first letter.



```
  SEND
+1ORE
-------------
1ONEY
```

Now in the column s10, s+1 ≥10. S must be 8 because there is a 1 carried over from the column EON or 9. O must be 0 (if s=8 and there is a 1 carried or s = 9 and there is no 1 carried) or 1 (if s=9 and there is a 1 carried). But 1 is already taken, so O must be 0.

There cannot be carry from column EON because any digit +0 < 10, unless there is a carry from the column NRE, and E=9; But this cannot be the case because then N would be 0 and 0 is already taken. So E < 9 and there is no carry from this column. Therefore S=9 because 9+1=10.



```
   9 E N D
+ 1O R E
-----------------
1ON E Y
```

In the column EON, E cannot be equal to N. So there must be carry from the column NRE; E+1=N. We now look at the column NRE, we know that E+1=N. Since we know that carry from this column, N+R=1E (if there is no carry from the column DEY) or N+R+1=1E (if there is a carry from the column DEY).

```
   9 E N D
 + 1 0 8 E
-----------------
 1 O N E Y
```

Now just think what are the digits we have left? They are 7, 6, 5, 4, 3 and 2. We know there must be a carry from the column DEY. So D + E % 10. N = E+ 1, SoE cannot be 7 because then N would be 8 which is already taken. D is almost 7, so E cannot be 2 because then $D + E < 10$ and E cannot be 3 because then $D + E = 10$ and $Y = 0$, but 0 is already taken. Also E cannot be 4 because if $D > 6, D + E < 10$ and if $D = 6$ or $D = 7$ then $Y = 0$ or $Y = 1$, which are both taken. So E is 5 or 6. If $E = 6$, then $D = 7$ and $Y = 3$. So this part will work but look the column N8E. Point that there is a carry from the column D5Y.$N + 8 + 1 = 16$(As there is a carry from this column). But then N=7 and 7 is taken by D therefore E=5.

```
    9 5 N D
  + 1 0 8 5
---------------------
  1 0 N 5 Y
```

Now we have gotten this important digit, it gets much simpler from here. N+8+1=15, N=6

```
    9 5 6 D
  + 1 0 8 5
--------------------
  1 0 6 5 Y
```

The digits left are 7, 4, 3 and 2. We know there is carry from the column D5Y, so the only pair that works is D=7 and Y= 2.

```
     9 5 6 7
  +  1 0 8 5
---------------------
  1  0 6 5 2
```

Which is final solution of the problem.

LOCAL SEARCH & METAHEURISTICS

The Local search algorithm operate using a single current state and generally move only to neighbours of that state. It is not systematic, they have two key advantages

1. They use very little memory- usually a constant amount
2. They can often find reasonable solutions in large or infinite state spaces for which systematic algorithms are unsuitable.

Heuristic methods

– "guided", but incomplete…

• To be used when search space is too big and cannot be searched exhaustively
• So-called ≪Metaheuristics≫ :
• general techniques to guide the search
• Experimented in various problems :
  – Traveling Salesman Problem (since 60 's)
  – scheduling, vehicle routing, cutting
• Simple but experimentally very efficient ...

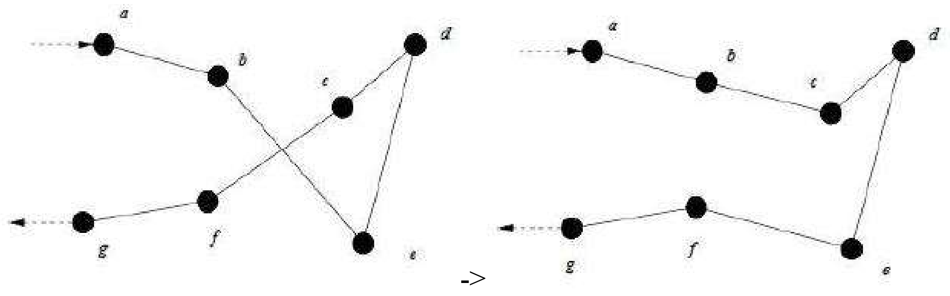
Travelling Sales Person(TSP) by Local Search

A *tour* can be represented by a permutation of the list of city nodes

• 2-opt: Swap the visit of 2 nodes

• Example:

(a, b, ***e,*** *d,* ***c,*** *f, g>⟶ <a, b,* ***c,*** *d,* ***e,*** *f, g>*



Naive Local Search algorithm

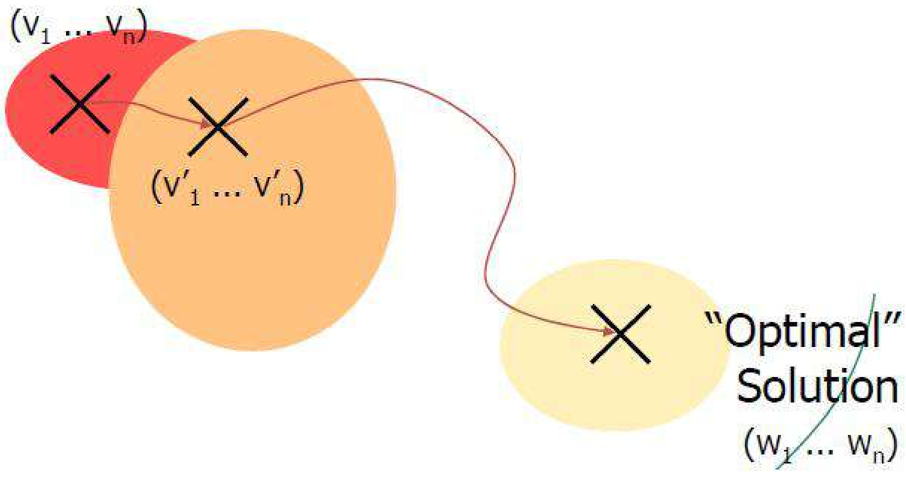
– Start by a random tour

– Consider all tours formed by executing swaps of 2 nodes

– Take the one with best (lower) cost

– Continue until optimum or time-limit reached

Local Search - Iterative Improvement



solving as optimization :

Optimization problem with objective function

– e.g. fitness function to maximize, or cost to minimize

• Basic algorithm :

– start from a random assignment

– Explore the ≪neighborhood≫

– move to a ≪ better ≫ candidate

– continue until optimal solution is found

• iterative improvement

• *anytime*algorithm

– outputs good if not optimal solution

<u>Hill-Climbing / Gradient Descent</u>

The Local search algorithm operate using a single current state and generally move only to neighbours of that state. It is not systematic, they have two key advantages

3. They use very little memory- usually a constant amount

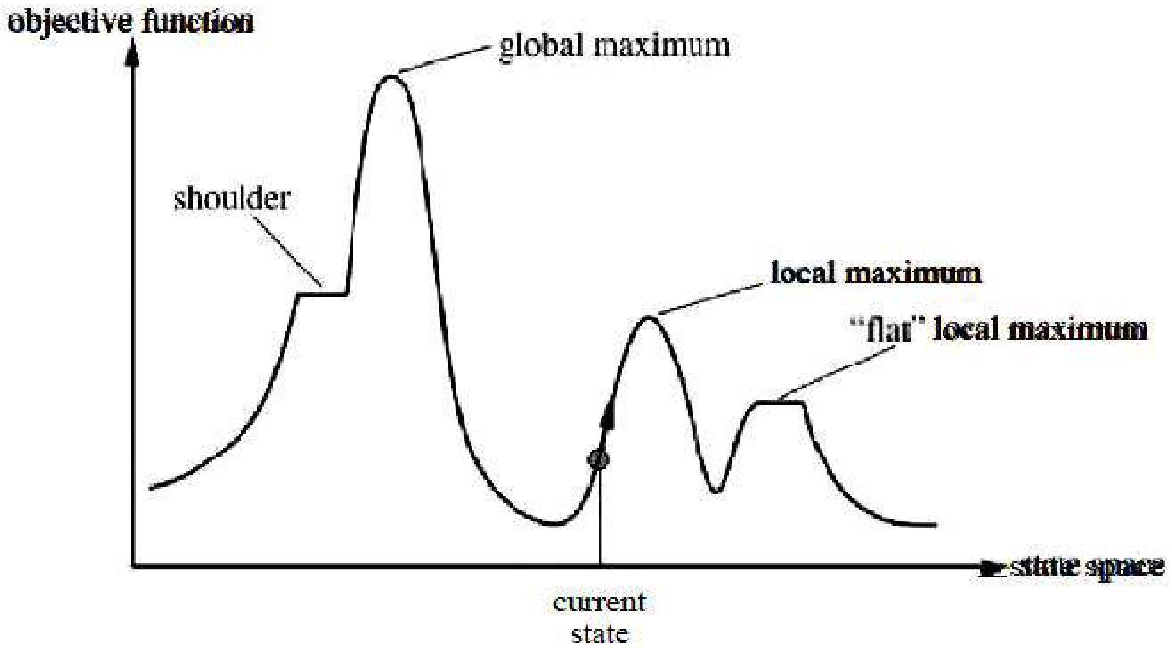
4. They can often find reasonable solutions in large or infinite state spaces for which systematic algorithms are unsuitable.

HILL CLIMBING

Hill climbing search algorithm is simply a loop that continuously moves in the direction of increasing value. It stops when it reaches a “peak” where no neighbour has higher value. This algorithm is considered to be one of the simplest procedures for implementing heuristic search. The hill climbing comes from that idea if you are trying to find the top of the hill and you go up direction from where ever you are. This heuristic combines the advantages of both depth first and breadth first searches into a single method.

Local Maxima:

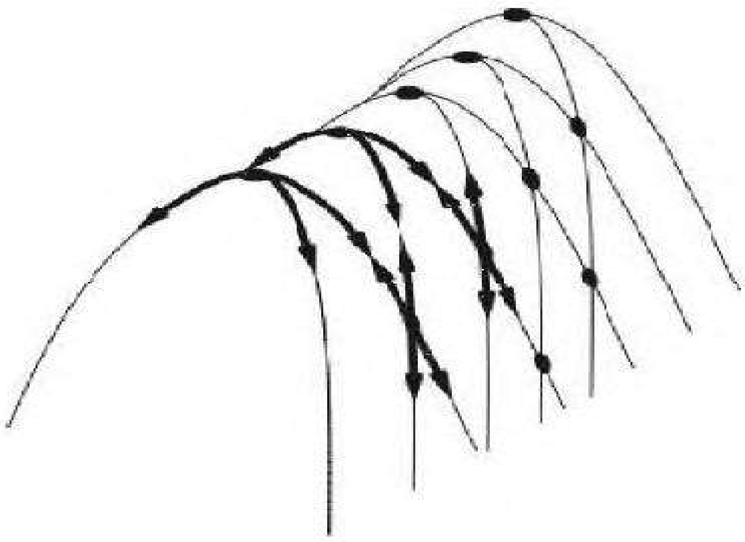
A local maxima is a state that is better than each of its neighbouring states, but not better than some other states further away.
Generally this state is lower than the global maximum. At this point, one cannot decide easily to move in which direction! This difficulties can be extracted by **the process of back tracking** i.e. backtrack to any of one earlier node position and try to go on a different event direction.
To implement this strategy, maintaining in a list of path almost taken and go back to one of them. If the path was taken that leads to a dead end, then go back to one of them.



Local Maxima

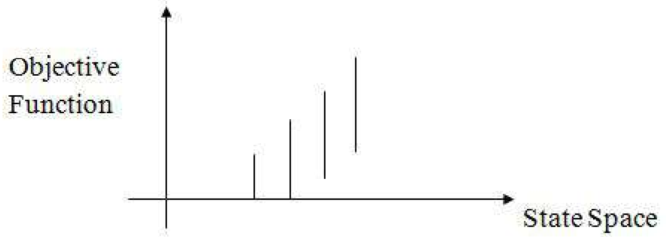
Ridges:

It is a special type of local maxima. It is a simply an area of search space. Ridges result in a **sequence of local maxima that is very difficult to implement ridge itself has a slope which is difficult to traverse.** In this type of situation **apply two or more rules before doing the tes**t. This will correspond to move inseveral directions at once.



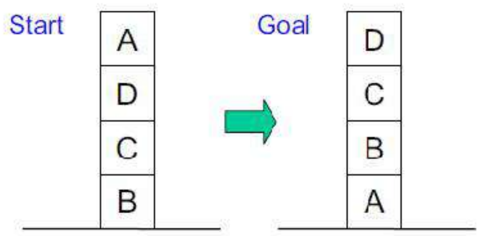
Plateau:

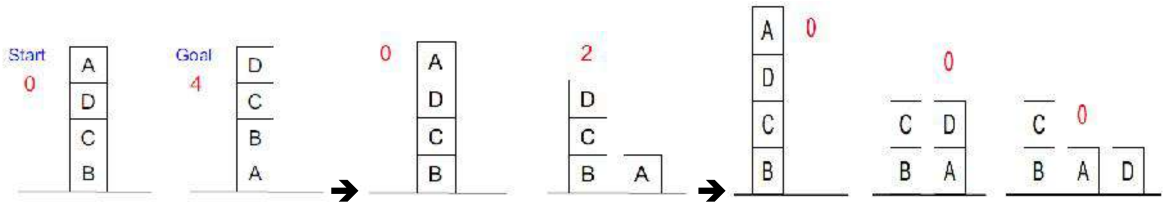
It is a flat area of search space in which **the neighbouring have same value**. So it is very difficult to calculate the best direction. So to get out of this situation, **make a big jump in any direction**, which will help to move in a new direction this is the best way to handle the problem like plateau.



- Hill climbing is a local method: Decides what to do next by looking only at the "immediate" consequences of its choices.
- Global information might be encoded in heuristic functions.
- Can be very inefficient in a large, rough problem space.
- Global heuristic may have to pay for computational complexity. Often useful when combined with other methods, getting it started right in the right general neighborhood.

<u>**Example :**</u>

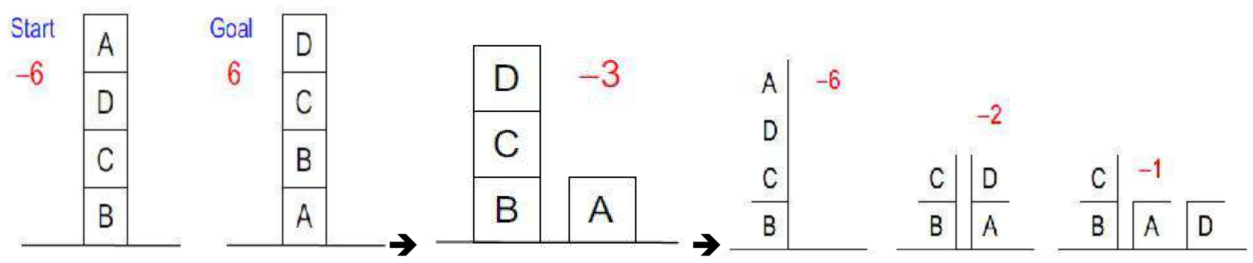




Local heuristic:

+1 for each block that is resting on the thing it is supposed to be resting on.

−1 for each block that is resting on a wrong thing.



Global heuristic:
For each block that has the correct support structure: +1 to every block in the support structure.
For each block that has a wrong support structure: −1 to every block in the support structure.
Can be very inefficient in a large, rough problem space.
• Global heuristic may have to pay for computational complexity.
• Often useful when combined with other methods, getting it started right in the right general neighbourhood.

UNIT I - QUESTIONS BANK
PART A

1. What is Artificial Intelligence? Nov/Dec2008

Artificial intelligence (AI) is a branch of computer science and engineering that deals with intelligent behavior, learning, and adaptation in machines.

- A branch of computer science that studies how to endow computers with capabilities of human intelligence.
- Examples include control, planning and scheduling, the ability to answer diagnostics and consumer questions handwriting, speech and facial recognition.

- Artificial Intelligence is the study of making

i. Systems that think like humans
ii. Systems that act like humans
iii. Systems that think rationally
iv. Systems that act rationally

2. What are the capabilities should the computer possess to pass the Turing Test?

- Natural Language Processing – To enable it to communicate successfully in English
- Knowledge Representation – to store what it knows or hears
- Automated Reasoning – to use the stored information to answer questions and to draw new conclusions
- Machine Learning – to adapt to new circumstances and to detect and extrapolate patterns

3. What is Turing Test?

- The Turing Test was proposed by ―Alan Turing ―.
- It was designed to provide a satisfactory operational definition of intelligence.
- This test is based on indistinguishability from undeniably intelligent entities – human beings.

4. What is Total Turing Test?

Total Turing Test includes a video signal so that the interrogator can test the subject's perceptual abilities , as well as the opportunity for the interrogator to pass physical objects ―through the hatch.

5. What are the capabilities should the computer possess to pass Total Turing Test?

Computer Vision – To perceive Objects
Robotics – to manipulate objects and move about.

6. What is percept and percept sequence of an agent?

- Percept is referred to the agent's perceptual inputs at any given instant.
- Percept sequence is the complete history of everything the agent has ever perceived.

7. What is an agent function and agent program ?

- An agent function describes the agent's behavior . It maps a given percept sequence to an action.
- An agent program implements the agent function mapping percepts to actions for an artificial agent

.

8. What is the difference between agent function and agent program ?

- **The agent function** is an abstract mathematical description . It takes the entire percept as input
- **The agent program** is a concrete implementation, running on the agent architecture. It takes the current percept as input

9. What are the four factors a rational agent depends?

- The performance measure that defines the criterion of success.
- The agent's prior knowledge of the environment
- The actions that the agent can perform
- The agent's percept sequence to date.

10. List the steps involved in simple problem solving agent.

(i) Goal formulation
(ii) Problem formulation
(iii) Search
(iv) Search Algorithm
(v) Execution phase

11. What is a task environment?

- A Task environment specification includes the

i. Performance measure
ii. The environment
iii. Agent's actuators
iv. Agent's sensors

- In designing an agent , the first step must always to specify the task environment as fully as possible

12. What is PEAS? Nov/Dec 2008

- P- Performance ,
- E- Environment
- A-Actuators
- S- Sensors
- These are the description of task environment of any problem.

13. Write short notes on performance measure.

- A performance measure evaluates the behavior of an agent in an environment.
- A performance measure embodies the criterion for success of an agent's behavior.
- When an agent is plunked down(dropped) , it generates a sequence of actions according to the percepts it receives. This sequence of actions causes the environment to go through a sequence of states.

14. Write the PEAS description of the task environment for an automated taxi.

- Agent type – Taxi driver
- Performance Measure – Safe, Fast, legal, comfortable trip, maximize profits.
- Environment – Roads, other traffic, Pedestrians, Customers
- Actuators – Steering, accelerator, brake, signal, horn, display
- Sensors- Cameras, Sonar, Speedometer, GPS, Odometer, Accelerometer

15. What are the properties of task environments?

- Fully observable or Partially observable
- Deterministic or stochastic
- Episodic or sequential
- Static or dynamic
- Discrete or continuous
- Single agent or multiagent

16. Give some examples of task environments.

- Crossword puzzle

- Chess with a clock
- Poker
- Taxi driving
- Medical diagnosis
- Image analysis

17. What are the components of well-defined problems?

- The **initial state** that the agent starts in . The initial state for our agent of example problem is described by *In(Arad)*
- A **Successor Function** returns the possible **actions** available to the agent. Given a state x, SUCCESSOR-FN(x) returns a set of {action, successor} ordered pairs where each action is one of the legal actions in state x, and each successor is a state that can be reached from x by applying the action.
  For example, from the state In(Arad),the successor function for the Romania
  problem would return
  { [Go(Sibiu),In(Sibiu)],[Go(Timisoara),In(Timisoara)],[Go(Zerind),In(Zerind)] }
- The **goal test** determines whether the given state is a goal state.
- A **path cost** function assigns numeric cost to each action. For the Romania problem the cost of path might be its length in kilometers.

18. What is an agent and rational agent? MAY/JUNE 2009

- **Agent** – An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators.
- Rational Agent – A rational agent is one that acts so as to achieve the best outcome or, when there is uncertainty , the best expected outcome.
- A rational agent chooses whichever action maximizes the expected value of the performance measure given the percept sequence to date

19. Write the structure of agents.

Agent = architecture + program

- Architecture- Computing device with physical sensors and actuators
- Agent Program – implements the agent function mapping percepts to actions.

20. What is problem formulation?

Problem formulation is the process of deciding what actions and states to consider , given a goal.

21. Give the four components used to define a problem

- Initial state
- Successor function
- Goal test
- Path cost

22. What is heuristic function?

- Heuristic function is a key component of Best-First-Search algorithms
- It is denoted by h(n).
- h(n)=estimated cost of the cheapest path from node n to goal node.

23. Define relaxed problem.

- A problem with fewer restrictions on the actions is called a relaxed problem.
- The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem.

24. Write the reasons hill climbing gets stuck.

- Local maxima

- Ridges
- Plateaux

25. Write about exploration problems.

Exploration problems arise when the agent has no idea about the states and actions of its environment.

26. Define CSP. Nov/Dec 2008

- CSP - Constraint Satisfaction Problem
- A CSP consists of variables with constraints on them.
- A CSP is defined by a set of variables , X1,X2,…Xn and a set of constraints , C1,C2,….Cm.
- Each variable Xi has a nonempty domain Di of possible values.
- Each constraint Ci involves some subset of the variables and specifies the allowable combinations of values for that subset.

27. Define the following: Local maxima ,Ridges ,Plateau

Local maxima : a local maximum is a peak that is higher than each of its neighboring states, but lower than the global maximum.

Ridges – Ridges result in a sequence of local maxima that is very difficult for greedy algorithms to navigate.

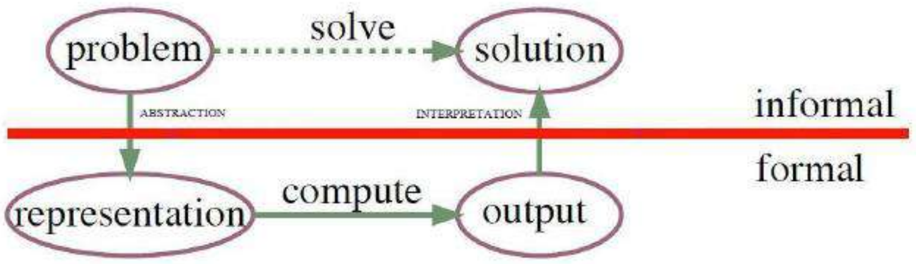
Plateau – A plateau is an area of the state space landscape where the evaluation function is flat.

PRATHYUSHA ENGINEERING COLLEGE
Department of Computer Science and Engineering
VI Semester (EVEN 2019-20)
CS8691ARTIFICIAL INTELLIGENCE
UNIT II

PROBLEM SOLVING METHODS 9

Problem solving Methods -Search Strategies-Uninformed -Informed -Heuristics -Local Search Algorithms and Optimization Problems -Searching with Partial Observations -Constraint Satisfaction Problems –Constraint Propagation -Backtracking Search -Game Playing -Optimal Decisions in Games –Alpha -Beta Pruning -Stochastic Games

1. PROBLEM SOLVING METHODS



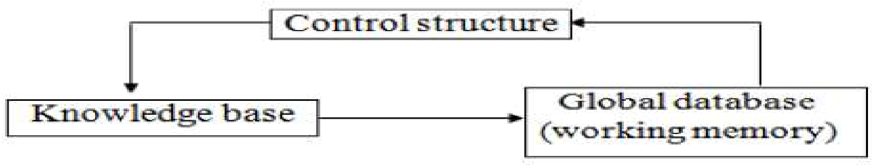
First, we must reduce it to one for which a precise statement can be given. This is done by defining the problem's state space, which includes the start and goal states and a set of operators for moving around in that space.

The problem can then be solved by **searching for a path through the space** from an initial state to a goal state. The process of solving the problem can usefully be modelled as a **production system**.

The production rules are also known as condition – action, antecedent – consequent, pattern – action, situation – response, feedback – result pairs.

For example,
If (you have an exam tomorrow) THEN (study the whole night)

The production system can be classified as monotonic, non-monotonic, partially commutative and commutative



In production system we have to choose the **appropriate control structure** so that the search can be as efficient as possible.

A production system is a system that adapts a system with production rules.
A *production system* consists of:

- **A set of production Rules**

A set of rules each consisting of a left side and a right hand side. Left hand side or pattern determines the applicability of the rule and a right side describes the operation to be performed if the rule is applied.

- **Knowledge/databases**

One or more knowledge/databases that contain whatever information is appropriate for the particular task. Some parts of the database may be **permanent**, while other parts of it may pertain only to the solution of the **current problem**. The information in these databases may be structured in any appropriate way.

- **Control strategy**

A control strategy that specifies the order in which the rules will be compared to the database and a way of resolving the conflicts that arise when several rules match at once.

- **A rule applied**

Production System also encompasses a family of general production system interpreters, including:

1. Basic production system languages, such as OPS5 and ACT*
2. More complex, often hybrid systems called *expert system shells,* which provide complete (relatively speaking) environments for the construction of knowledge-based expert systems**.**
3. General problem-solving architectures like SOAR, a system based on a specific set of cognitively motivated hypotheses about the nature of problem solving.

Features of Production System

Some of the main features of production system are:

1. **Expressiveness and intuitiveness** : In real world, many times situation comes like "if this happen-you will do that", "if this is so-then this should happen" and many more. The production rules essentially tell us what to do in a given situation.
2. **Simplicity**:
   The structure of each sentence in a production system is unique and uniform as they use "IF-THEN" structure. This structure provides simplicity in knowledge representation. This feature of production system improves the readability of production rules.
3. **Modularity**:
   This means production rule code the knowledge available in discrete pieces.
   Information can be treated as a collection of independent facts which may be added or deleted from the system with essentially no deleterious side effects.
4. **Modifiability**:
   This means the facility of modifying rules. It allows the development of production rules in a skeletal form first and then it is accurate to suit a specific application.
5. **Knowledge intensive:**
   The knowledge base of production system stores pure knowledge. This part does not contain any type of control or programming information. Each production rule is normally written as an English sentence; the problem of semantics is solved by the very structure of the representation.

Disadvantages of production system

1.**Opacity**:

This problem is generated by the combination of production rules. The opacity is generated because of less prioritization of rules. More priority to a rule has the less opacity.

2.**Inefficiency**:

During execution of a program several rules may active. A well devised control strategy reduces this problem. As the rules of the production system are large in number and they are hardly written in hierarchical manner, it requires some forms of complex search through all the production rules for each cycle of control program.

3.**Absence of learning:**

Rule based production systems do not store the result of the problem for future use. Hence, it does not exhibit any type of learning capabilities. So for each time for a particular problem, some new solutions may come.

4.**Conflict resolution**:

The rules in a production system should not have any type of conflict operations. When a new rule is added to a database, it should ensure that it does not have any conflicts with the existing rules.

- **Graph Search**
  - – Representation of states and transitions/actionsbetween states → graph
  - – Explored explicitly or implicitly
- **Constraint Solving**
  - – Represent problem by variables and constraints
  - – Use specific solving algorithms to speedup search
- **Local Search and Metaheuristics**
  - – Evaluation function to check if state is "good" or not
  - – Optimization of the evaluation function

<u>GRAPH SEARCH</u>

- A large variety of problems can berepresented by a graph
- A (directed) graph consists of a set of nodes and a set of directed arcs between nodes.
- The idea is to find a path along these arcs from a start node to a goal node.
- A directed **graph** consists of
- a set *N* of **nodes** and
- a set *A* of ordered pairs of nodes called **arcs**.

A **path** from node *s* to node *g* is a sequence of nodes $\langle n_0, n_1, ..., n_k \rangle$

A **tree** is

- a DAG where there is one node with no incoming arcs and every other node has exactly one incoming arc.
- Node with no incoming arcs is called the **root** of the tree
- Nodes with no outgoing arcs are called **leaves**.

To encode problems as graphs,

- One set of nodes is referred to as the **start nodes**
- Another set is called the **goal nodes**.

A **solution** is a path from a start node to a goal node.

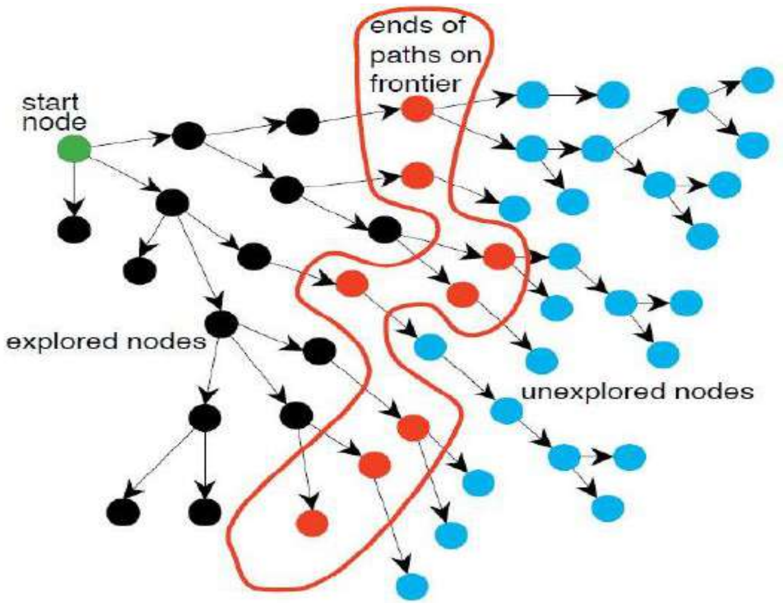
An **optimal solution** is one of the least-cost solutions; that is, it is a path *p* from a start node to a goal node such that there is no path *p'* from a start node to a goal node where *cost(p')<cost(p)*.
The **forward branching factor** of a node is the number of arcs leaving the node.
The **backward branching factor** of a node is the number of arcs entering the node. These factors provide measures of the complexity of graphs.

• Solutions can be considered as definingspecific nodes
• Solving the problem is reduced to searchingthe graph for those nodes
  – starting from an initial node
  – each transition in the graph corresponds to apossible action
  – ending when reaching a final node (solution)

Defining Search Problems

- A statement of a Search problem has 4 components
  1. A set of states
  2. A set of "operators" which allow one to get from one state to another
  3. A start state S
  4. A set of possible goal states, or ways to test for goal states

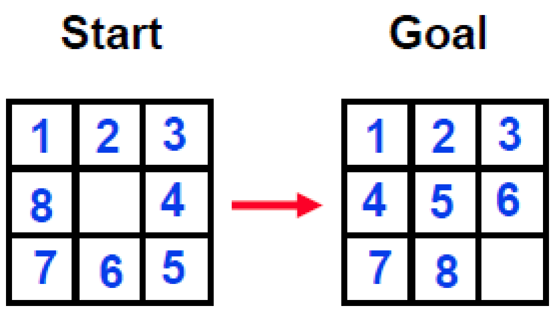
  4a. Cost path
- Search solution consists ofa sequence of operators which transform S into a goal state G



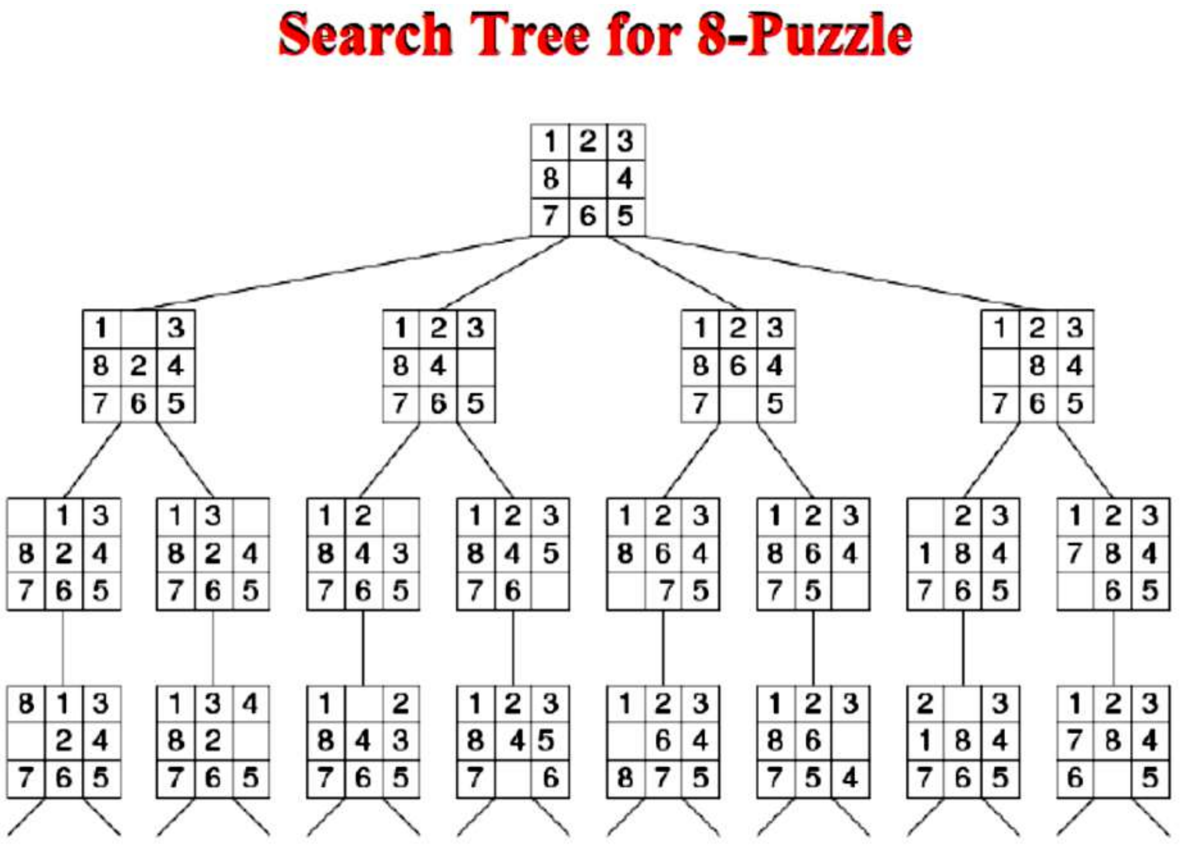
BASIC GRAPH SEARCH ALGORITHM

```
Input: a graph,
        a set of start nodes,
        Boolean procedure $goal(n)$ that tests if $n$ is a goal node.
$frontier := \{\langle s \rangle : s$ is a start node$\}$;
while $frontier$ is not empty:
        select and remove path $\langle n_0, \ldots, n_k \rangle$ from $frontier$;
        if $goal(n_k)$
            return $\langle n_0, \ldots, n_k \rangle$;
        for every neighbor $n$ of $n_k$
            add $\langle n_0, \ldots, n_k, n \rangle$ to $frontier$;
end while
```

The 8-puzzle



• can be generalized to15-puzzle, 24-puzzle, etc.
• Any (n2 -1)-puzzle for n ≥ 3
• state = permutation of (0, 1, 2, 3, 4, 5, 6, 7, 8)
  Eg.state above is: (1,2,3,8,0,4,7,6,5)
• 9! = 362,880 possible states
• Solution: (1,2,3,4,5,6,7,8,0)
• Actions: possible moves



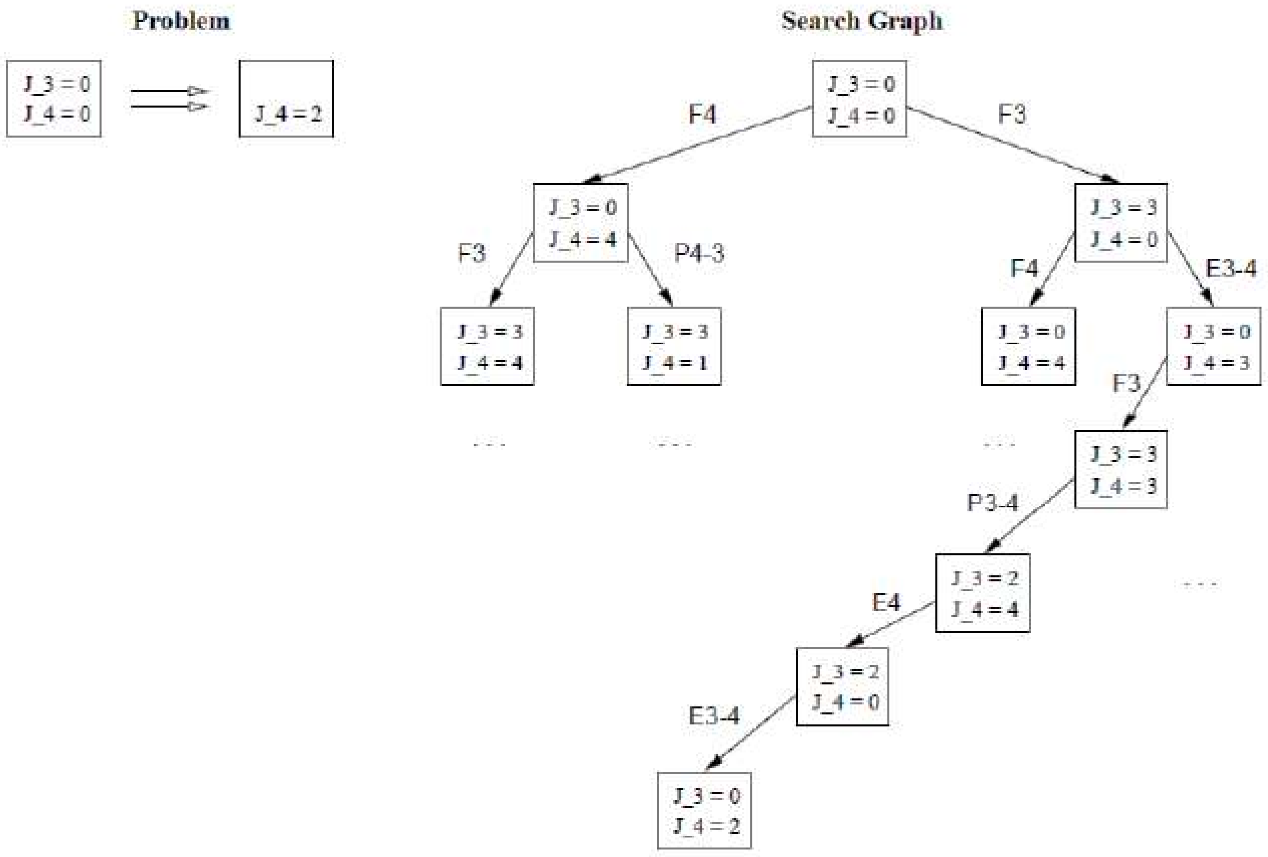
Search:
•Expand out possible nodes
•Maintain a fringe of as yet unexpanded nodes
•Try to expand as few tree nodes as possible

Water Jug Problem

A Water Jug Problem: You are given two jugs, a 4-gallon one and a 3-gallon one, a pump which has unlimited water which you can use to fill the jug, and the ground on which water may be poured. Neither jug has any measuring markings on it. How can you get exactly 2 gallons of water in the 4-gallon jug?

State Representation and Initial State–we will represent a state of the problem as a tuple (x,y) where x represents the amount of water in the 4-gallon jug and y represents the amount of water in the 3-gallon jug. Note $0 \leq x \leq 4$, and $0 \leq y \leq 3$. Our initial state: (0,0)

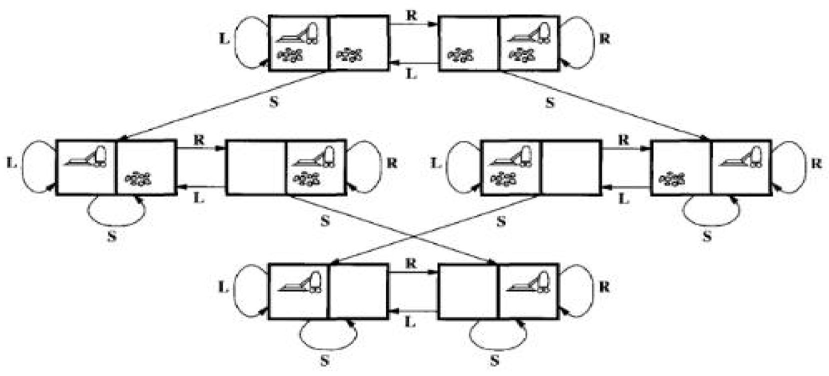
•State: (x,y) x=0,1,2,3,or 4 y =0,1,2,3
•Start state: (0,0).
•Goal state: (2,n) for any n.
•Attempting to end up in a goal state



TOY PROBLEMS
Vacuum World Example

o States: The agent is in one of two locations.,each of which might or might not contain dirt. Thus there are 2 x 22 = 8 possible world states.
o Initial state: Any state can be designated as initial state.
o Successor function : This generates the legal states that results from trying the three actions (left, right, suck). The complete state space is shown in figure 2.3
o Goal Test : This tests whether all the squares are clean.
o Path test : Each step costs one ,so that the the path cost is the number of steps in the path.

Vacuum World State Space



The state space for the vacuum world.
Arcs denote actions: L = Left,R = Right,S = Suck

<u>CONSTRAINT MODELING</u>

Search can be made easier in cases where the solution of corresponding to an optimal path, is only required to satisfy local consistency conditions. We call such problems *Constraint Satisfaction (CS) Problems.*

Many problems can be stated as constraints satisfaction problems.

Two-step process:

1. Constraints are discovered and propagated as far as possible.
2. If there is still not a solution, then search begins, adding new constraints.

Two kinds of rules:

1. Rules that define valid constraint propagation.
2. Rules that suggest guesses when necessary.

.

- **Declarative language** : modeling is easy
- **local** specification of the problem
- **global** consistency achieved (or approximated)by constraint solving techniques
- **compositionality** : constraints are combined implicitly through shared logical variables

Basic Objects

Variable : a place holder for values *X,Y, Z, L ,U , List* 3 21

Function Symbol: mapping of variables to values : Sin(x), Cos(x), x+y, x-y

Relation Symbol:

Relation between variables

Arithmetic relation : +, -, x, /

Symbolic relation : *all_different*

Constraints

Declarative relations between variables

• Constraints used to both model and solve the problem

• specific algorithms for efficient computation Constraints

• Constraints could be numeric or symbolic :

$X \leq 5$, $X + Y = Z$
all_different(X1,X2,…,Xn)
at_most(N,[X1,X2,X3],V)

• multi-directional relations

Crypto-arithmetics as CSP

```
   S E N D
+ M O R E
_________
M O N E Y
```

each letter represents a (different) digit
and the addition should be correct !
... Solution ?

```
C4 C3 C2 C1
     S   E  N  D
+   M   O  R  E
_______________
M  O  N  E  Y
```

variables :{S,E,N,D,M,O,R,Y,R1, R2, R3, R4}
domains :{0,...,9} for the letters, {0,1} for the carries
constraints : all_different(S,E,N,D,M,O,R,Y} $S \neq 0$ $M \neq 0$

We have to replace each letter by a distinct digit so that the resulting sum is correct.
In the above problem, M must be 1. You can visualize that, this is an addition problem. The sum of two four digit numbers cannot be more than 10,000. Also M cannot be zero according to the rules, since it is the first letter.

```
   SEN D
 +1 0 R E
--------------
 1 0 N E Y
```

Now in the column s10, $s+1 \geq 10$. S must be 8 because there is a 1 carried over from the column EON or 9. O must be 0 (if s=8 and there is a 1 carried or s = 9 and there is no 1 carried) or 1 (if s=9 and there is a 1 carried). But 1 is already taken, so O must be 0.

There cannot be carry from column EON because any digit +0 < 10, unless there is a carry from the column NRE, and E=9; But this cannot be the case because then N would be 0 and 0 is already taken. So E < 9 and there is no carry from this column. Therefore S=9 because 9+1=10.

```
    9 E N D
 + 1 O R E
-----------------
  1 O N E Y
```

In the column EON, E cannot be equal to N. So there must be carry from the column NRE; E+1=N. We now look at the column NRE, we know that E+1=N. Since we know that carry from this column, N+R=1E (if there is no carry from the column DEY) or N+R+1=1E (if there is a carry from the column DEY).

```
   9 E N D
 + 1 0 8 E
-----------------
 1 O N E Y
```

Now just think what are the digits we have left? They are 7, 6, 5, 4, 3 and 2. We know there must be a carry from the column DEY. So D + E % 10. N = E+ 1, SoE cannot be 7 because then N would be 8 which is already taken. D is almost 7, so E cannot be 2 because then $D + E < 10$ and E cannot be 3 because then $D + E = 10$ and $Y = 0$, but 0 is already taken. Also E cannot be 4 because if $D > 6, D + E < 10$ and if $D = 6$ or $D = 7$ then $Y = 0$ or $Y = 1$, which are both taken. So E is 5 or 6. If $E = 6$, then $D = 7$ and $Y = 3$. So this part will work but look the column N8E. Point that there is a carry from the column D5Y.$N + 8 + 1 = 16$(As there is a carry from this column). But then N=7 and 7 is taken by D therefore E=5.

```
   9 5 N D
 + 1 0 8 5
-------------------
 1 0 N 5 Y
```

Now we have gotten this important digit, it gets much simpler from here. N+8+1=15, N=6

```
   9 5 6 D
 + 1 0 8 5
-------------------
 1 0 6 5 Y
```

The digits left are 7, 4, 3 and 2. We know there is carry from the column D5Y, so the only pair that works is D=7 and Y= 2.

```
    9 5 6 7
 +  1 0 8 5
--------------------
 1  0 6 5 2        Which is final solution of the problem.
```

2. SEARCH STRATEGIES

Problem solving in artificial intelligence may be characterized as a systematic search through a range of possible actions in order to reach some predefined goal or solution. In AI problem solving by search algorithms is quite common technique. In the comingage of AI it will have big impact on the technologies of the robotics and path finding. It is also widely used in travel planning. A search algorithm takes a problem as input and returns the solution in the form of an action sequence.

Once the solution is found, the actions it recommends can be carried out. This phase is called as the execution phase. After formulating a goal and problem to solve the agent cells a search procedure to solve it .

A search strategy is defined by picking the order of node expansion

• Strategies are evaluated along the following dimensions:

– completeness: does it always find a solution if one exists?

– time complexity: number of nodes generated
– space complexity: maximum number of nodes in memory
– optimality: does it always find a least-cost solution?
– systematicity: does it visit each state at most once?

DIFFERENT TYPES OF SEARCHING

The searching algorithms can be various types. When any type of searching is performed, there may some information about the searching or mayn't be. Also it is possible that the searching procedure may depend upon any constraints or rules. However, generally searching can be classified into two types i.e. uninformed searching and informed searching. Also some other classifications of these searches are

Search Algorithms
- Uninformed Search
  - DFS
  - BFS
  - Iterative Deepening DFS
  - Uniform cost search
  - Bidirectional Search
  - Branch and bound search
- Informed Search
  - Generate and test Search
  - Best First Search
  - A*
  - AO*
  - Greedy Search

- **Strategies for search**

  Some widely used control strategies for search are stated below**.**

  - **Forward search :** Here, the control strategies for exploring search proceeds forward from initial state towards a solution; the methods are called *data-directed*.
  - **Backward search :** Here, the control strategies for exploring searchproceeds backward from a goal or final state towards either a solvable sub problem or the initial state; the methods are called *goal directed***.**
  - **Both forward and backward search :** Here, the control strategies for exploring search is a *mixture of both* forward and backward strategies .
  - **Systematic search :** Where search space is small, a systematic (but blind) method can be used to explore the whole search space.

    One such search method is *depth-first search* and the other is *breath-first search*.

3. UNINFORMED SEARCH (Blind Search)

- Breadth First Search (BFS) – Refer previous section
- Depth First Search (DFS) – Refer previous section
- Depth-limited search (DLS)
- Iterative deepening depth first search
- Uniform-cost search (UCS)

Breadth First Search

Breadth first search is a general technique of traversing a graph. Breadth first search may use more memory but will always find the shortest path first. In this type of search the state space is represented inform of a tree.

The solution is obtained by traversing through the tree. The nodes of the tree represent the start value or starting state, various intermediate states and the final state. In this search a queue datastructure is used and it is level by level traversal. Breadth first search expands nodes in order of their distance from the root. It is a path finding algorithm that is capable of always finding the solution if oneexists.

The solution which is found is always the optional solution. This task is completed in a very memory intensive manner. Each node in the search tree is expanded in a breadth wise at each level.

Concept:

Step 1: Traverse the root node

Step 2: Traverse all neighbours of root node.

Step 3: Traverse all neighbours of neighbours of the root node.

Step 4: This process will continue until we are getting the goal node.

Algorithm: Breadth-First Search

```
1. Create a variable called NODE-LIST and set it to the initial state.
2. Until a goal state is found or NODE-LIST is empty:
   (a) Remove the first element from NODE-LIST and call it E. If NODE-LIST was empty, quit.
   (b) For each way that each rule can match the state described in E do:
        (i) Apply the rule to generate a new state.
       (ii) If the new state is a goal state, quit and return this state.
      (iii) Otherwise, add the new state to the end of NODE-LIST.
```

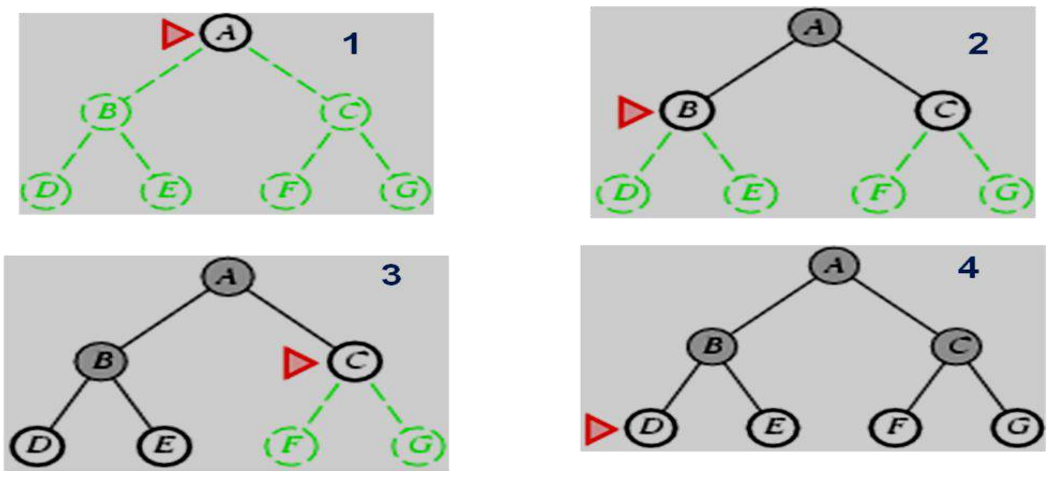
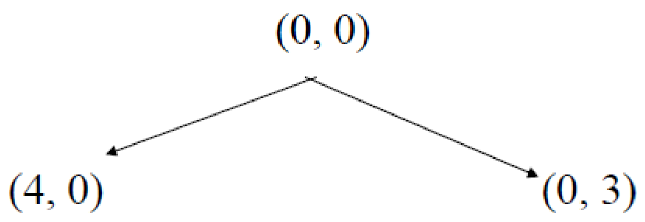


Figure : Search tree of Breadth first search

Example :

- Construct a tree with the initial state as its root.
- Generate all the offspring of the root by applying each of the applicable rules to the initial state



- Now for each leaf node, generate all its successors by applying all the rules that are appropriate
- Continue this process until some rule produces a goal state.

Evaluation

– Complete? - **Yes (b is finite)**
– Time Complexity? **O(b^d)**
– Space Complexity? **O(b^d)**
– Optimal? **Yes, if stepcost=1**

Advantages:

In this procedure at any way it will find the goal.
It does not follow a single unfruitful path for a long time.
It finds the minimal solution in case of multiple paths.

Disadvantages:

BFS consumes large memory space.
Its time complexity is more.
It has long pathways, when all paths to a destination are on approximately the same search depth.

DEPTH FIRST SEARCH (DFS)

DFS is also an important type of uniform search. DFS visits all the vertices in the graph. This type of algorithm always chooses to go deeper into the graph. After DFS visited all the reachable vertices from a particular sources vertices it chooses one of the remaining undiscovered vertices and continues the search. DFS reminds the space limitation of breath first search by always generating next a child of the deepest unexpanded nodded. The data structure stack or last in first out (LIFO) is used for DFS. One interesting property of DFS is that, the discover and finish time of each vertex from a parenthesis structure. If we use one open parenthesis when a vertex is finished then the result is properly nested set of parenthesis.

Concept:

Step 1: Traverse the root node.
Step 2: Traverse any neighbour of the root node.
Step 3: Traverse any neighbour of neighbour of the root node.
Step 4: This process will continue until we are getting the goal node.

Algorithm:

1.If the initial state is a goal state, quit and return success.
2.Otherwise, loop until success or failure is signalled.
a) Generate a state, say E, and let it be the successor of the initial state. If there is no successor, signal failure.

b) Call Depth-First Search with E as the initial state.

c) If success is returned, signal success. Otherwise continue in this loop.

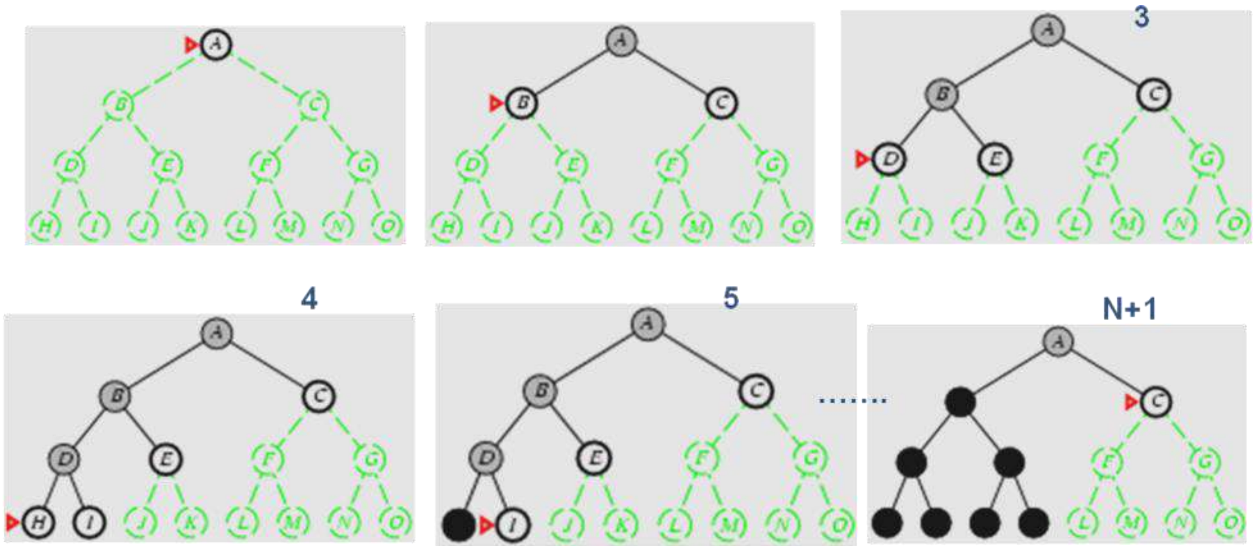


Figure : Search tree for depth first search

Implementation:

Let us take an example for implementing the above DFS algorithm.

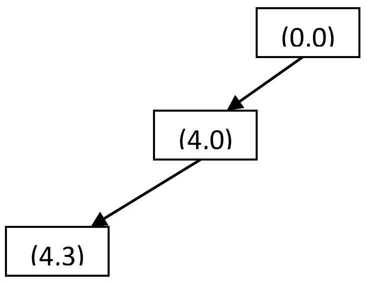


Figure Examples of DFS

Evaluation

– Complete? - **No**

– Time Complexity?- **O(b^m)**

– Space Complexity? - **O(bm)**

Advantages:

- DFS consumes very less memory space.
- It will reach at the goal node in a less time period than BFS if it traverses in a right path.
- It may find a solution without examining much of search because we may get the desired solutionin the very first go.

Disadvantages:

- It is possible that may states keep reoccurring.
- There is no guarantee of finding the goal node.
- Sometimes the states may also enter into infinite loops.

Difference between BFS and DFS

BFS

1. It uses the data structure queue.
2. BFS is complete because it finds the solution if one exists.
3. BFS takes more space i.e. equivalent to o (b0) where b is the maximum breath exist in a search tree and d is the maximum depth exit in a search tree.
4. In case of several goals, it finds the best one.

DFS

1. It uses the data structure stack.
2. It is not complete because it may take infinite loop to reach at the goal node.
3. The space complexity is O (d).
4. In case of several goals, it will terminate the solution in any order.

<u>DEPTH-LIMITED SEARCH (DLS)</u>

- <u>**Description:**</u>

  We can avoid examining unbounded branches by limiting the search to depth l. The nodes at level l are treaded as if they have no successors. We will call depth limit *l*, this solves the infinite path problem.

- <u>**Performance Measure:**</u>

  <u>Completeness:</u>

  The limited path introduces another problem which is the case when we choose $l < d$, in which is our DLS will never reach a goal, in this case we can say that DLS is not complete.

  <u>Optimality:</u>

  One can view DFS as a special case of the depth DLS, that DFS is DLS with l = infinity.

  DLS is not optimal even if $l > d$.

  <u>Time Complexity:</u> $O(b^l)$

  <u>Space Complexity:</u> $O(b^l)$

ITERATIVE DEEPENING DEPTH FIRST SEARCH (IDDFS)

One way to combine the space efficiency of depth-first search with the optimality of breadth-first methods is to use **iterative deepening**. The idea is to recompute the elements of the frontier rather than storing them. Each recomputation can be a depth-first search, which thus uses less space.

Consider making a breadth-first search into an iterative deepening search. This is carried out by having a depth-first searcher, which searches only to a limited depth. It can first do a depth-first search to depth 1 by building paths of length 1 in a depth-first manner. Then it can build paths to depth 2, then depth 3, and so on. It can throw away all of the previous computation each time and start again. Eventually it will find a solution if one exists, and, as it is enumerating paths in order, the path with the fewest arcs will always be found first.

When implementing an iterative deepening search, you have to distinguish between

- failure because the depth bound was reached and

- failure that does not involve reaching the depth bound.

In the first case, the search must be retried with a larger depth bound. In the second case, it is a waste of time to try again with a larger depth bound, because no path exists no matter what the depth. We say that failure due to reaching the depth bound is **failing unnaturally**, and failure without reaching the depth bound is **failing naturally**.

```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution, or fail-
ure
    inputs: problem, a problem
    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH( problem, depth)
        if result ≠ cutoff then return result
```

The iterative-deepening search fails whenever the breadth-first search would fail. When asked for multiple answers, it only returns each successful path once, even though it may be rediscovered in subsequent iterations. Halting is achieved by keeping track of when increasing the bound could help find an answer:

- The depth bound is increased if the depth bound search was truncated by reaching the depth bound. In this case, the search failed *unnaturally*. The search failed *naturally* if the search did not prune any paths due to the depth bound. In this case, the program can stop and report no (more) paths.
- The search only reports a solution path if that path would not have been reported in the previous iteration. Thus, it only reports paths whose length is the depth bound.

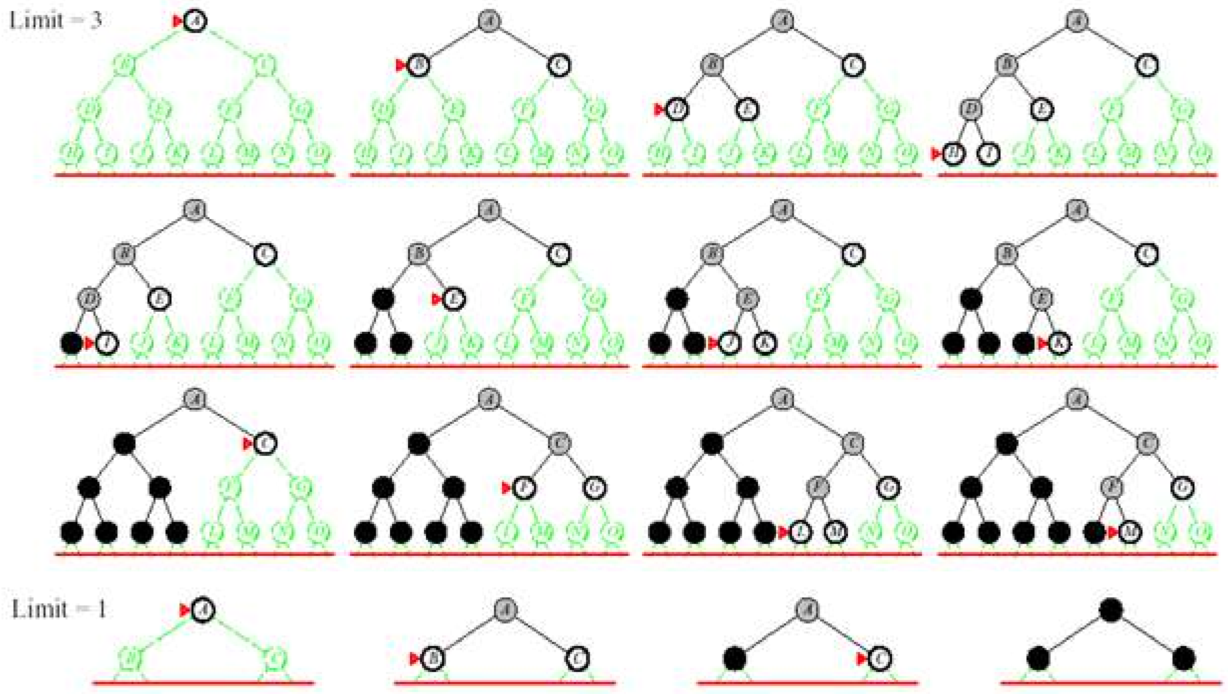
The obvious problem with iterative deepening is the wasted computation that occurs at each step. This, however, may not be as bad as one might think, particularly if the branching factor is high. Consider the running time of the algorithm. Assume a constant branching factor of *$b>1$*. Consider the search where the bound is *k*. At depth *k*, there are b^k nodes; each of these has been generated once. The nodes at depth *$k-1$* have been generated twice, those at depth *$k-2$* have been generated three times, and so on, and the nodes at depth *1* have been generated *k* times. Thus, the total number of nodes generated is

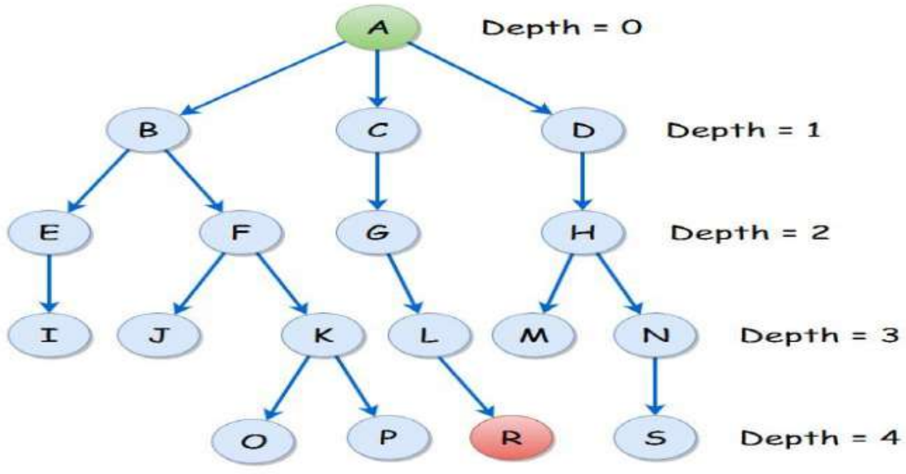
$b^k+2b^{k-1}+3b^{k-}$
$+\cdots+kb$

$$= b^k(1+2b^{-1}+3b^{-2}+\cdots+kb^{1-k})$$
$$\leq b^k(\textstyle\sum_{i=1}^{\infty} ib^{(1-i)})$$
$$= b^k(b/(b-1))^2.$$

As there are *$(b/(b-1))\ b^k$* nodes in the tree, iterative deepening has a constant overhead of *$(b/(b-1))$* times the cost of generating the nodes at depth *n*. When *$b=2$* there is an overhead factor of 2, and when *$b=3$* there is an overhead factor of *1.5* over generating the frontier. This algorithm is *$O(\ b^k)$* and there cannot be an asymptotically better uninformed search strategy. Note that, if the branching factor is close to 1, this analysis does not work (because then the denominator would be close to zero).

Iterative deepening can also be applied to an A^* search. **Iterative deepening A^*** (IDA*) performs repeated depth-bounded depth-first searches. Instead of the bound being on the number of arcs in the path, it is a bound on the value of *f(n)*. The threshold starts at the value of *f(s)*, where *s* is the starting node with minimal *h*-value. IDA* then carries out a depth-first depth-bounded search but never expands a node with a higher *f*-value than the current bound. If the depth-bounded search fails *unnaturally*, the next bound is the minimum of the *f*-values that exceeded the previous bound. IDA* thus checks the same nodes as A^* but recomputes them with a depth-first search instead of storing them.





Our starting node (A) is at a depth of 0. Our goal node (R) is at a depth of 4. The above example is a finite tree, but think of the above tree as an infinitely long tree and only up to depth = 4 is shown in the diagram.

In IDDFS, we perform DFS up to a certain depth and keep incrementing this allowed depth. Performing DFS upto a certain allowed depth is called **Depth Limited Search** (DLS). As Depth Limited Search (DLS) is important for IDDFS.

Let us understand DLS, by performing DLS on the above example. In Depth Limited Search, we first set a constraint on how deep (or how far from root) will we go. Let's say our limit (*DEPTH*) is 2. Now, in the above diagram, place your hand to cover the nodes at depth 3 and 4. Now, by looking at the rest of the nodes, can you tell the order in which a normal DFS would visit them? It would be as follows –

```
1A B E F C G D H
```

Can you do it for *DEPTH* = {0, 1, 2, 3, 4} ? Just cover the nodes you don't need with your hand and try to perform DFS in you mind. You should get answers like this –

DEPTH	DLS traversal
0	A
1	A B C D
2	A B E F C G D H
3	A B E I F J K C G L D H M N
4	A B E I F J K O P C G L R D H M N S

Properties of Iterative deepening Search

Space complexity = O(bd)

Time Complexity	$B+(b+b^2)+ \dots (b+\dots b^d) = O(b^d)$
Complete?	Yes
Optimal	Only if path cost is a non-decreasing function of depth

IDS combines the small memory footprint of DFS, and has the completeness guarantee of BFS

UNIFORM-COST SEARCH (UCS)

- **Description:**

  Uniform-cost is guided by path cost rather than path length like in BFS, the algorithms starts by expanding the root, then expanding the node with the lowest cost from the root, the search continues in this manner for all nodes.

  where there is no goal state and processing continues until the shortest path to all nodes has been determined.

- **Performance Measure:**

  Completeness:

  It is obvious that UCS is complete if the cost of each step exceeds some small positive integer, this to prevent infinite loops.

  Optimality:

  UCS is always optimal in the sense that the node that it always expands is the node with the least path cost.

  Time Complexity:

  UCS is guided by path cost rather than path length so it is hard to determine its complexity in terms of b and d, so if we consider C to be the cost of the optimal solution, and every action costs at least e, then the algorithm worst case is $O(b^{C/e})$.

  Space Complexity:

  The space complexity is $O(b^{C/e})$ as the time complexity of UCS.

BIDIRECTIONAL SEARCH

Description:

If node predecessors are easy to compute – eg pred(n) = S(n)- then search can proceed simultaneously forward from the initial state and backward from the goal state, those 2 searches stop when they meet each other at some point in the middle of the graph. The motivation for this idea is that $2b^{d/2} << b^d$

The following pictures illustrates a bidirectional search

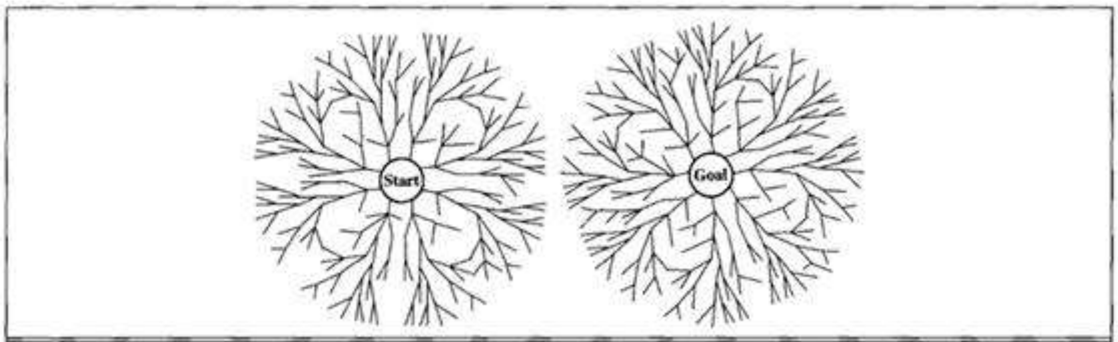


Figure A schematic view of a bidirectional search that is about to succeed, when a branch from the start node meets a branch from the goal node.

- **Performance Measure:**

Completeness:

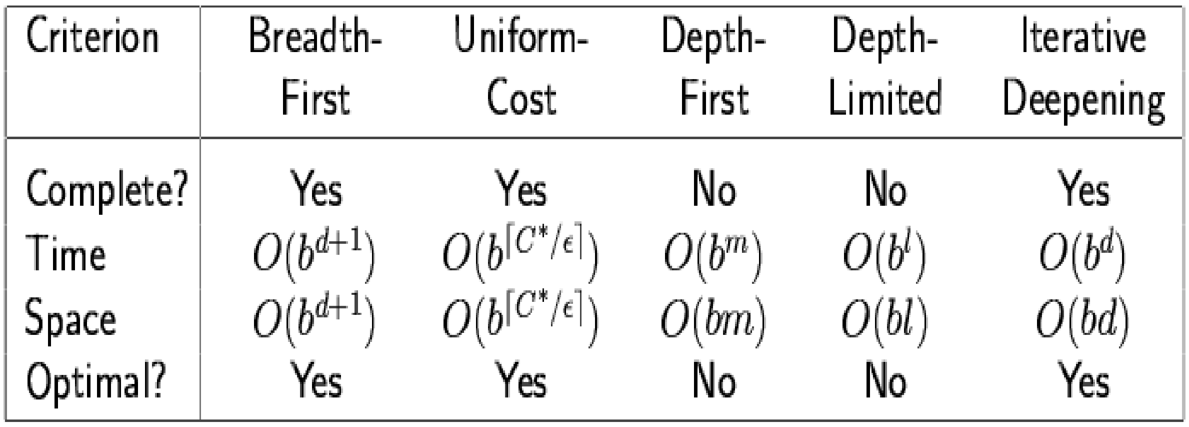
Bidirectional search is complete when we use BFS in both searches, the search that starts from the initial state and the other from the goal state.

Optimality:

- Like the completeness, bidirectional search is optimal when BFS is used and paths are of a uniform cost – all steps of the same cost.
- Other search strategies can be used like DFS, but this will sacrifice the optimality and completeness, any other combination than BFS may lead to a sacrifice in optimality or completeness or may be both of them.

Time and Space Complexity:

- May be the most attractive thing in bidirectional search is its performance, because both searches will run the same amount of time meeting in the middle of the graph, thus each search expands $O(b^{d/2})$ node, in total both searches expand $O(b^{d/2} + b^{d/2})$ node which is too far better than the $O(b^{d+1})$ of BFS.
- If a problem with b = 10, has a solution at depth d = 6, and each direction runs with BFS, then at the worst case they meet at depth d = 3, yielding 22200 nodes compared with 11111100 for a standard BFS.
- We can say that the time and space complexity of bidirectional search is $O(b^{d/2})$.



Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes	Yes	No	No	Yes
Time	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal?	Yes	Yes	No	No	Yes

4. INFORMED SEARCH (HEURISTIC SEARCH)

Heuristic is a technique which makes our search algorithm more efficient. Some heuristics help to guide asearch process without sacrificing any claim to completeness and some sacrificing it.

Heuristic is a problem specific knowledge that decreases expected search efforts. It is a technique which sometimes works but not always. **Heuristic search algorithm uses information about the problem to help directing thepath through the search space**. These searches uses some functions that estimate the cost from the currentstate to the goal presuming that such function is efficient. A heuristic function is a function that maps from problem state descriptions to measure of desirability usually represented as number. The purpose of heuristic function is to guide the search process in the most profitable directions by suggesting which path to follow first when more than is available.

Generally heuristic incorporates domain knowledge to improve efficiency over blind search. In AI heuristic has a general meaning and also a more specialized technical meaning. Generally a term heuristicis used for any advice that is effective but is not guaranteed to work in every case.

state descriptions →measures of desirability

For example in case of travelling sales man (TSP) problem we are using a heuristic to calculate the nearest neighbour. Heuristic is a method that provides a better guess about the correct choice to make at any junction that would beachieved by random guessing. This technique is useful in solving though problems which could not besolved in any other way. Solutions take an infinite time to compute.

Let us see some classifications of heuristic search.

1. **Generate and Test Search**
2. **Best-first Search**
3. **A* Search**
4. **Constraint Search**
5. **Greedy Search**
6. **Means-ends analysis**

Generate and Test Search

Generate-and-test search algorithm is a very simple algorithm that guarantees to find a solution if done systematically and there exists a solution.

1. Generate a possible solution.
2. Test to see if this is actually a solution.
3. Quit if a solution has been found.
   Otherwise, return to step 1.

Potential solutions that need to be generated vary depending on the kinds of problems. For some problems the possible solutions may be particular points in the problem space and for some problems, paths from the start state.

Generate-and-test, like depth-first search, requires that complete solutions be generated for testing. In its most systematic form, it is only an exhaustive search of the problem space. Solutions can also be generated randomly but solution is not guaranteed. This approach is what is known as British Museum algorithm: finding an object in the British Museum by wandering randomly.

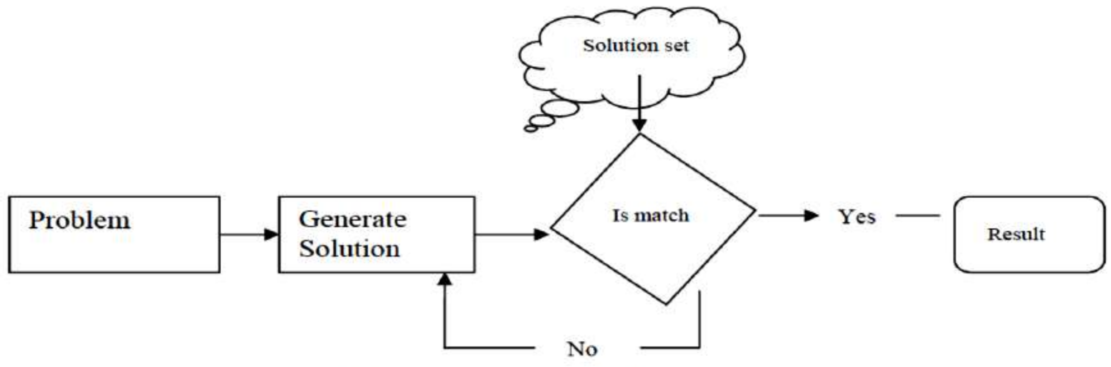


Figure 7 Working of Generate and test search algorithm

While generating complete solutions and generating random solutions are the two extremes there exists another approach that lies in between. The approach is that the search process proceeds systematically but some paths that unlikely to lead the solution are not considered. This evaluation is performed by a heuristic function.

Depth-first search tree with backtracking can be used to implement systematic generate-and-test procedure. As per this procedure, if some intermediate states are likely to appear often in the tree, it would be better to modify that procedure to traverse a graph rather than a tree.

Advantages: Acceptable for simple problems.
Disadvantage : Inefficient for problems with large space.

• Exhaustive generate-and-test.
• Heuristic generate-and-test: not consider paths that seem unlikely to lead to a solution.
• Plan generate-test:
  − Create a list of candidates.
  − Apply generate-and-test to that list.

Example:
“Arrange four 6-sided cubes in a row, with each side of each cube painted one of four colours, such that on all four sides of the row one block face of each colour is showing.”

Heuristic: if there are more red faces than other colours then, when placing a block with several red faces, use few of them as possible as outside faces.

GREEDY BEST – FIRST SEARCH

- Greedy best-first search tries to expand the node that is closest to the goal, on the grounds that this is likely to lead to a solution quickly

• Thus, the evaluation function is f(n) = h(n)
• E.g. in minimizing road distances a heuristic lower bound for distances of cities is their straight-line distance
• Greedy search ignores the cost of the path that has already been traversed to reach n
• Therefore, the solution given is not necessarily optimal
• If repeating states are not detected, greedy best-first search may oscillate forever between two promising states

- Because greedy best-first search can start down an infinite path and never return to try other possibilities, it is incomplete
- Because of its greediness the search makes choices that can lead to a dead end; then one backs up in the search tree to the deepest unexpanded node
- Greedy best-first search resembles depth-first search in the way it prefers to follow a single path all the way to the goal, but will back up when it hits a dead end
- The worst-case time and space complexity is $O(b^m)$
- The quality of the heuristic function determines the practical usability of greedy search
-

GREEDY : Best First Search
Combining the two is to follow a single path at a time, but switch paths whenever some competing path looks more promising than the current one.

- Best first search is an instance of graph search algorithm in which a node is selected for expansion based on evaluation function f (n).
- The node which is the lowest evaluation is selected for the explanation because the evaluation measures distance to the goal.
- Best first search can be implemented within general search frame work via a priority queue, a data structure that will maintain the fringe in ascending order of f values.
- This search algorithm **serves as combination of depth first and breadth first search algorithm**. Best first search algorithm is often referred greedy algorithm this is because they quickly attack the most desirable path as soon as its heuristic weight becomes the most desirable.
- For both depth-first and breadth-first search, which node in the search tree will be considered next only depends on the structure of the tree.
- • The rationale in best-first search is to **expand those paths next that seem the most "promising"**. Making this vague idea of what may be promising precise means defining heuristics.
- Heuristics by means of a heuristic function h that is used to estimate the "distance" of the current node n to a goal node:
  h(n) = estimated cost from node n to a goal node

The definition of h is highly application-dependent. In the route-planning domain, for instance, we could use the straight-line distance to the goal location. For the eight-puzzle, we might use the number of misplaced tiles.

Concept:

Step 1: Traverse the root node

Step 2: Traverse any neighbour of the root node, that is maintaining a least distance from the root node and insert them in ascending order into the queue.

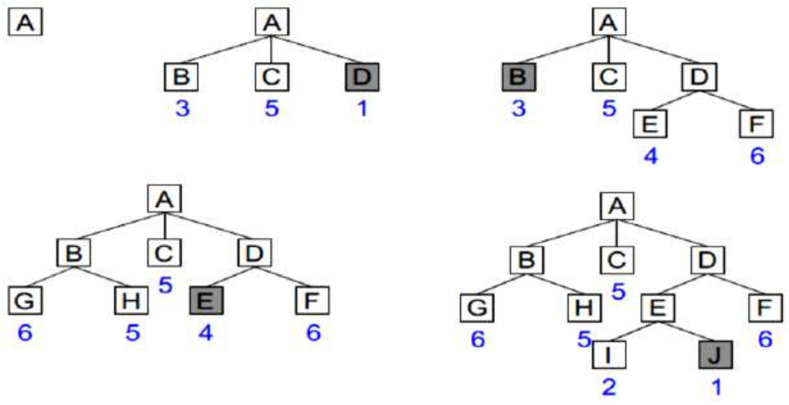
Step 3: Traverse any neighbour of neighbour of the root node, that is maintaining a least distance from the root node and insert them in ascending order into the queue

Step 4: This process will continue until we are getting the goal node

- OPEN: nodes that have been generated, but have not examined. This is organized as a priority queue.
- CLOSED: nodes that have already been examined. Whenever a new node is generated, check whether it has been generated before.

Algorithm

```
1. OPEN = {initial state}.
2. Loop until a goal is found or there are no nodes left in OPEN:
        − Pick the best node in OPEN
        − Generate its successors
        − For each successor:
        new→ evaluate it, add it to OPEN, record its parent
        generated before → change parent, update successors
```



<u>Greedy search:</u>

h(n) = cost of the cheapest path from node n to agoal state.

Neither optimal nor complete

<u>• Uniform-cost search:</u>

g(n) = cost of the cheapest path from the initial state to node n.

Optimal and complete, but very inefficient

Advantage:

- It is more efficient than that of BFS and DFS.

- Time complexity of Best first search is much less than Breadth first search.
- The Best first search allows us to switch between paths by gaining the benefits of both breadth first and depth first search. Because, depth first is good because a solution can be found without computing all nodes and Breadth first search is good because it does not get trapped in deadends.

Disadvantages:
Sometimes, it covers more distance than our consideration.

A* SEARCH

- A* is a cornerstone name of many AI systems and has been used since it was developed in 1968 by PeterHart; Nils Nilsson and Bertram Raphael.
- It is the combination of Dijkstra's algorithm and Best first search. It can be used to solve many kinds of problems.
- A* search finds the shortest path through a search space to goal state using heuristic function. This technique finds minimal cost solutions and is directed to a goal state called A* search.
- In A*, the * is written for optimality purpose.
- The A* algorithm combines features of uniform-cost search and pure heuristic search to efficiently compute optimal solutions.

The A* algorithm also **finds the lowest cost path between the start and goal state**, where changing from one state to another requires some cost.

A* requires **heuristic function to evaluate the cost of path** that passes through the particular state. This algorithm is complete if the branching factor is finite and every action has fixed cost. It can be defined by following formula.

$$f(n) = g(n) + h(n)$$

Where

g(n): The actual cost path from the *start state to the current state*.
h (n): The estimated cost path from the *current state to goal state*.
f (n): The cost path from the start state to the goal state.

If *h(n)* is an underestimate of the path costs from node *n* to a goal node, then *f(p)* is an underestimate of a path cost of going from a start node to a goal node via *p*.

Function f*(n):At any node n,it is the actual cost of an optimal path from node s to node n plus the cost of an optimal path from node n to a goal node.
$f^*(n)=g^*(n)+h^*(n)$
Where $g^*(n)$=cost of the optimal path in the search tree from s to n;
$h^*(n)$=cost of the optimal path in the search tree from n to a goal node;

For the implementation of A* algorithm we will use two arrays namely OPEN and CLOSE.
OPEN: An array which contains the nodes that has been generated but has not been yet examined.
CLOSE: An array which contains the nodes that have been examined.

Algorithm:
1.Create a *search graph* G,consisting solely of the start node s. Puts on a list called OPEN.

2 .Create a list called CLOSED that is initially empty.

3.LOOP: if OPEN is empty, exit with failure.

4. Select the first node on OPEN, remove it from OPEN and put it on CLOSED. Call this node n.

5.If n is a goal node,exit successfully with the solution obtained by tracing a path along the pointers from n to s in G.

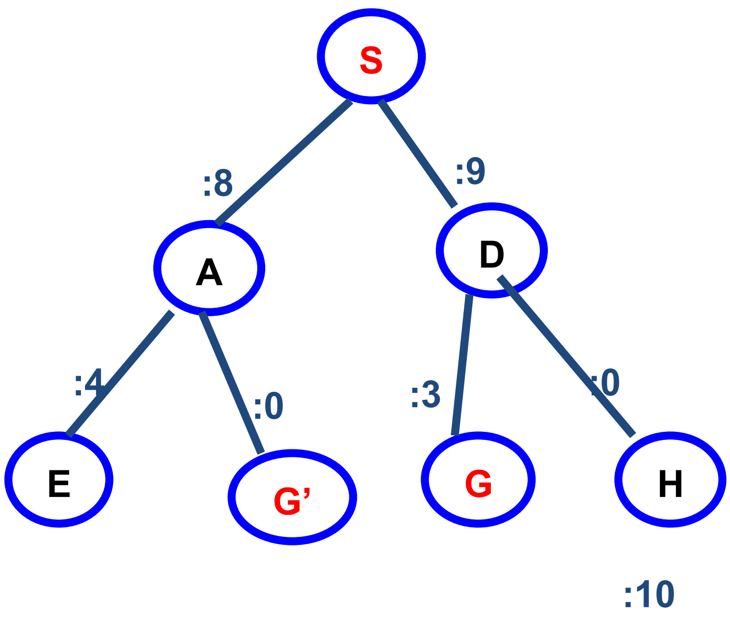
6.Expand node n, generating the set, M, of its successors and install them as successors of n in G.

7.Establish a pointer to n from those members of M that were not already in G(I .e, not already on either OPEN or CLOSED). Add these members of M to OPEN.For each member of M that was already on OPEN or CLOSED,decide whether or not to redirect its pointer to n. For each member of M already on CLOSED, decide for each of its descendents in G whether or not to redirect its pointer.

8.Reorder the list OPEN, either according to some scheme or some heuristic merit.

9.Goto LOOP

Example :



Path cost for S-D-G

f(S) = g(S) + h(S)

= 0 + 10 $\rightarrow$ **10**

f(D) = (0+3) + 9 $\rightarrow$ ***12***

f(G) = (0+3+3) + 0 $\rightarrow$ ***6***

Total path cost = *f(S)+f(D)+f(G)* $\rightarrow$ ***28***

Path cost for S-A-G'

f(S) = 0 + 10 $\rightarrow$ **10**

f(A) = (0+2) + 8 $\rightarrow$ ***10***

f(G') = (0+2+2) + 0 $\rightarrow$ ***4***

Total path cost = *f(S)+f(A)+f(G')* $\rightarrow$ ***24***

* Path S-A-G is chosen = Lowest cost

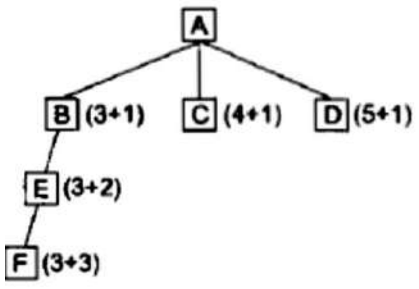


Fig. 3.4 *h' Underestimates h*

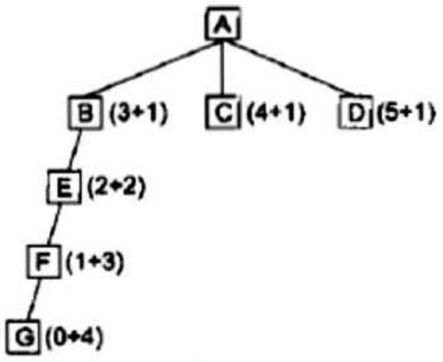


Fig. 3.5 *h' Overestimates h*

Implementation:
The implementation of A* algorithm is 8-puzzle game.

Advantages:

- It is complete and optimal.
- It is the best one from other techniques.
- It is used to solve very complex problems.
- It is optimally efficient, i.e. there is no other optimal algorithm guaranteed to expand fewer nodes than A*.

Disadvantages:

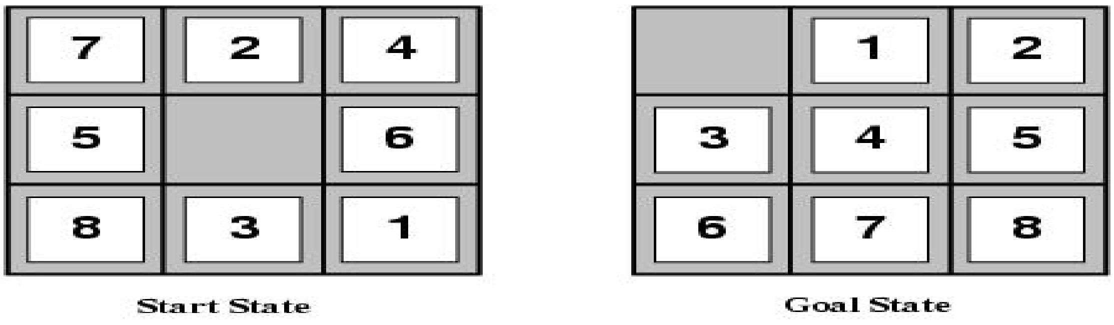
- This algorithm is complete if the branching factor is finite and every action has fixed cost.
- The speed execution of A* search is highly dependant on the accuracy of the heuristic algorithm that is used to compute h (n).
- It has complexity problems.

Admissibility
The property that A* always finds an optimal path, if one exists, and that the first path found to a goal is optimal is called the **admissibility** of A*. Admissibility means that, even when the search space is infinite, if solutions exist, a solution will be found and the first path found will be an optimal solution - a lowest-cost path from a start node to a goal node.

Example heuristic functions:
• Let h1(n) be the straight-line distance to the goal location. This is an admissible heuristic, because no solution path will ever be shorter than the straight-line connection.

5. HEURISTIC FUNCTIONS

- A heuristic function or simply a heuristic is a function that ranks alternatives in various search algorithms at each branching step basing on an available information in order to make a decision which branch is to be followed during a search



A typical instance of the 8-puzzle.

The solution is 26 steps long.

The 8-puzzle

- The 8-puzzle is an example of Heuristic search problem.
- The object of the puzzle is to slide the tiles horizontally or vertically into the empty space until the configuration matches the goal configuration
- The average cost for a randomly generated 8-puzzle instance is about 22 steps.
- The branching factor is about 3.(When the empty tile is in the middle,there are four possible moves;when it is in the corner there are two;and when it is along an edge there are three).
- This means that an exhaustive search to depth 22 would look at about 322 approximately = X 1010 states.
- By keeping track of repeated states, we could cut this down by a factor of about 170, 000, because there are only 9!/2 = 181,440 distinct states that are reachable.
- This is a manageable number, but the corresponding number for the 15-puzzle is roughly
- 1013.
- If we want to find the shortest solutions by using A*,we need a heuristic function that never overestimates the number of steps to the goal.
- The two commonly used heuristic functions for the 15-puzzle are : (1) h1 = the number of misplaced tiles.
- In the above figure all of the eight tiles are out of position,
  so the start state would have h1 = 8. h1 is an admissible heuristic.

(2) h2 = the sum of the distances of the tiles from their goal positions. This is called the city block distance or Manhattan distance.

h2 is admissible ,because all any move can do is move one tile one step closer to the **goal.**

Tiles 1 to 8 in start state give a Manhattan distance of

h2 = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18.

Neither of these overestimates the true solution cost, which is 26.

Inventing admissible heuristic functions

Relaxed problems

o A problem with fewer restrictions on the actions is called a ***relaxed problem***

o The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem

o If the rules of the 8-puzzle are relaxed so that a tile can move *anywhere*, then *h1(n)* gives the shortest solution

o If the rules are relaxed so that a tile can move to *any adjacent square,* then *h2(n)* gives the shortest solution

<u>6. LOCAL SEARCH & META HEURISTICS</u>

The Local search algorithm operate using a single current state and generally move only to neighbours of that state. It is not systematic, they have two key advantages

1. They use very little memory- usually a constant amount
2. They can often find reasonable solutions in large or infinite state spaces for which systematic algorithms are unsuitable.

Heuristic methods

– "guided", but incomplete…

• To be used when search space is too big and cannot be searched exhaustively

• So-called ≪Metaheuristics≫ :

• general techniques to guide the search

• Experimented in various problems :

– Traveling Salesman Problem (since 60 's)

– scheduling, vehicle routing, cutting

• Simple but experimentally very efficient ...

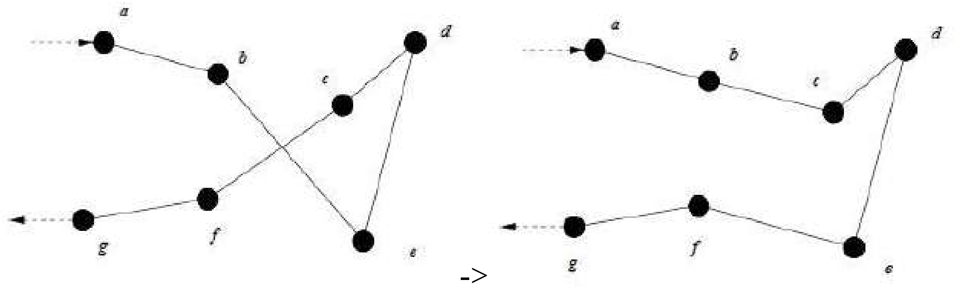
Travelling Sales Person(TSP) by Local Search

A *tour* can be represented by a permutation of the list of city nodes

• 2-opt: Swap the visit of 2 nodes

• Example:

(a, b, **e,** *d,* **c,** *f, g>→ <a, b,* **c,** *d,* **e,** *f, g>*



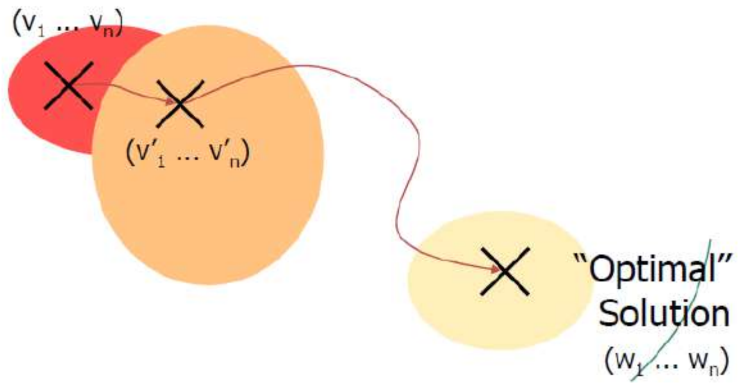
Naive Local Search algorithm

– Start by a random tour

– Consider all tours formed by executing swaps of 2 nodes

– Take the one with best (lower) cost
– Continue until optimum or time-limit reached

Local Search - Iterative Improvement



solving as optimization :
Optimization problem with objective function
– e.g. fitness function to maximize, or cost to minimize
• Basic algorithm :
– start from a random assignment
– Explore the ≪neighborhood≫
– move to a ≪ better ≫ candidate
– continue until optimal solution is found
• iterative improvement
• *anytime*algorithm
– outputs good if not optimal solution

<u>Why Hill-Climbing / Gradient Descent</u>

The Local search algorithm operate using a single current state and generally move only to neighbours of that state. It is not systematic, they have two key advantages

1. They use very little memory- usually a constant amount
2. They can often find reasonable solutions in large or infinite state spaces for which systematic algorithms are unsuitable.

HILL CLIMBING

Hill climbing search algorithm is simply a loop that continuously moves in the direction of increasing value. It stops when it reaches a "peak" where no neighbour has higher value. This algorithm is considered to be one of the simplest procedures for implementing heuristic search. The hill climbing comes from that idea if you are trying to find the top of the hill and you go up direction from where ever you are. This heuristic combines the advantages of both depth first and breadth first searches into a single method.

Hill climbing search

- It is simply a loop that continually moves in the direction of increasing value – that is uphill. It terminates when it reaches a "peak" where no neighbor has a higher value.

- The algorithm does not maintain a search tree, so the current node data structure need only record the state and its objective function value. Hill-climbing does not look ahead beyond the immediate neighbors of the current state. This resembles trying to find the top of Mount Everest in a thick fog while suffering from amnesia.

Hill climbing search algorithm

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
  inputs: problem, a problem
  local variables: current, a node
                   neighbor, a node

  current ← MAKE-NODE(INITIAL-STATE[problem])
  loop do
      neighbor ← a highest-valued successor of current
      if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
      current ← neighbor
```

- Local search algorithms typically use a complete-state formulation, where each state has 8 queens on the board, one per column. The successor function returns all possible states generated by moving a single queen to another square in the same column (so each state has 8 X 7 = 56 successors)

- The heuristic cost function h is the number of pairs of queens that are attacking each other; either directly or indirectly.

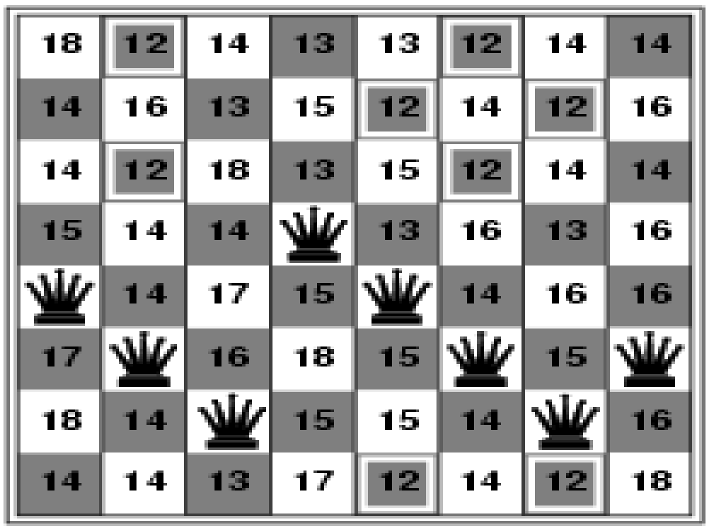
- The global minimum of this function is zero, which occurs only at perfect solutions.

- The figure shows a state with h = 17. The figure also shows the values of all its successors with the best successors having h = 12.

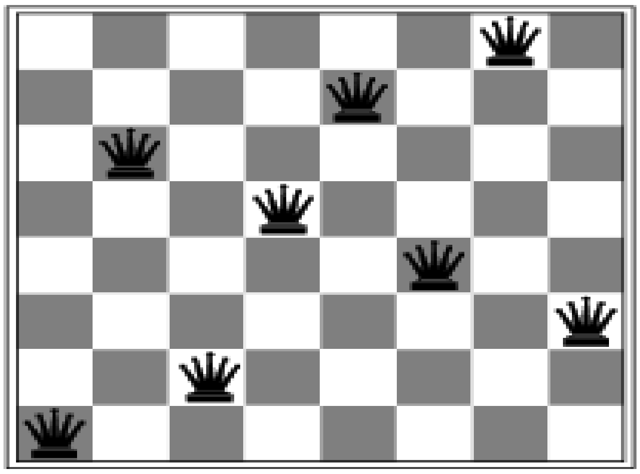
- Hill-climbing typically chooses randomly among the set of best successor, if there is more than one.

- Hill-climbing is sometimes called greedy local search because it grabs a good neighbor state without thinking ahead about where to go next. It often makes very rapid progress towards a solution, because it is usually quite easy to improve a bad state.

Figure: An 8-queens state with heuristic cost estimate h = 17, showing the value of h for each possible successor obtained by moving a queen within its column. The best moves are marked.



A Local minimum in the 8 –queens state space; the stat has h = 1 but every successor has a higher cost



For example, from state in figure (a), it takes just five steps to reach the state in figure(b), which has h = 1 and is very nearly a solution.

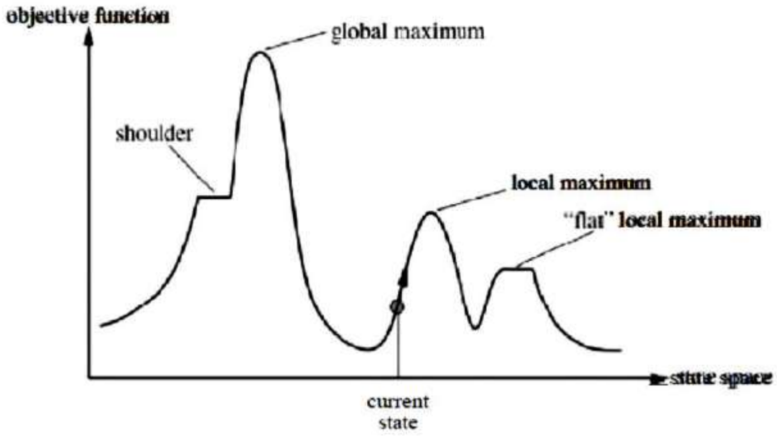
Unfortunately, hill climbing often gets stuck for the following reasons.

Local Maxima:

A local maxima is a state that is better than each of its neighbouring states, but not better than some other states further away.

Generally this state is lower than the global maximum. At this point, one cannot decide easily to move in which direction! This difficulties can be extracted by **the process of back tracking** i.e. backtrack to any of one earlier node position and try to go on a different event direction.

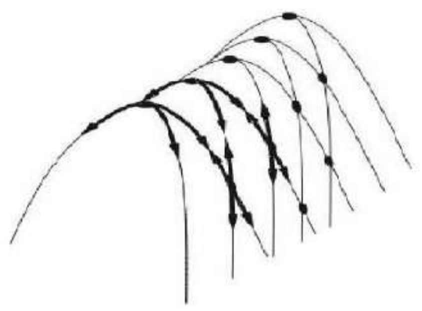
To implement this strategy, maintaining in a list of path almost taken and go back to one of them. If the path was taken that leads to a dead end, then go back to one of them.



Local Maxima

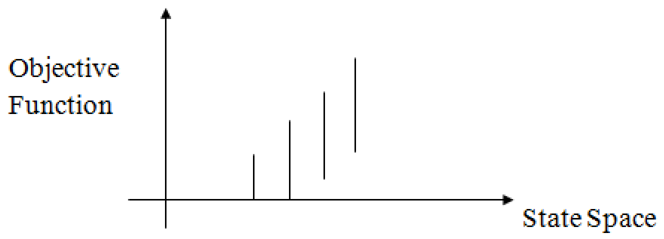
Ridges:

It is a special type of local maxima. It is a simply an area of search space. Ridges result in a **sequence of local maxima that is very difficult to implement ridge itself has a slope which is difficult to traverse.** In this type of situation **apply two or more rules before doing the tes**t. This will correspond to move inseveral directions at once.



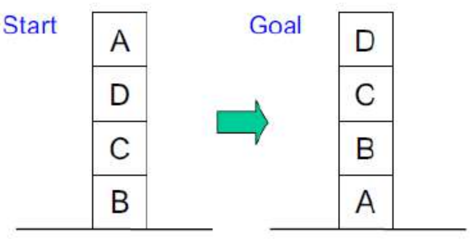
Plateau:

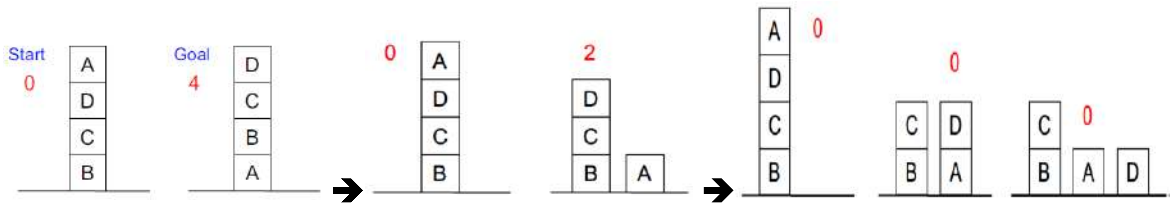
It is a flat area of search space in which **the neighbouring have same value**. So it is very difficult to calculate the best direction. So to get out of this situation, **make a big jump in any direction**, which will help to move in a new direction this is the best way to handle the problem like plateau.



- Hill climbing is a local method: Decides what to do next by looking only at the "immediate" consequences of its choices.
- Global information might be encoded in heuristic functions.
- Can be very inefficient in a large, rough problem space.
- Global heuristic may have to pay for computational complexity. Often useful when combined with other methods, getting it started right in the right general neighborhood.

<u>Example :</u>

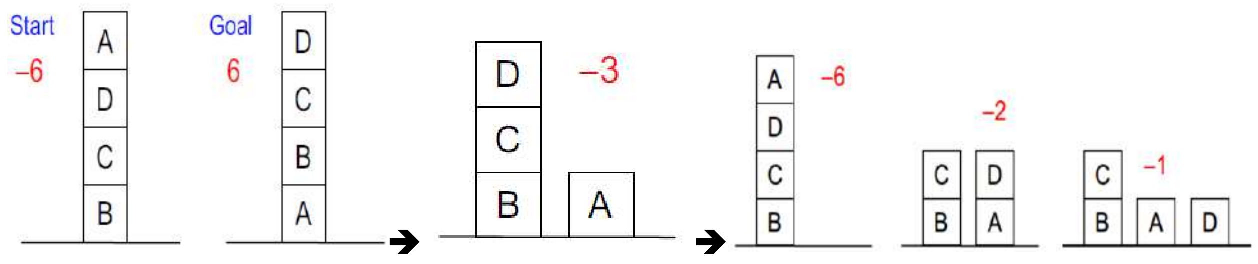




Local heuristic:

+1 for each block that is resting on the thing it is supposed to be resting on.

−1 for each block that is resting on a wrong thing.



Global heuristic:
For each block that has the correct support structure: +1 to every block in the support structure.
For each block that has a wrong support structure: −1 to every block in the support structure.
Can be very inefficient in a large, rough problem space.
• Global heuristic may have to pay for computational complexity.
• Often useful when combined with other methods, getting it started right in the right general neighbourhood.

7. ONLINE SEARCH AGENTS AND UNKNOWN ENVIRONMENT

- Offline search algorithms compute a complete solution before setting foot in the real world and then execute the solution without recourse to their percepts.
- Online search agent operates by interleaving computation and action : first it takes an action, then it observes the environment and computes the next action.
- Online search is an even better idea for stochastic domains.
- In general, an offline search would have to come up with an exponentially large contingency plan that considers all possible happenings, while an online search need only consider what actually does happen
- For example, a chess playing agent is well-advised to make its first move long before it has figured out the complete course of the game.
- Online search is a necessary idea for an **exploration problem**, where the states and actions are unknown to the agent.
- An agent in this state of ignorance must use its actions as experiments to determine what to do next, and hence must interleave computation and action.
- The canonical example of online search is a robot that is placed in a new building and must explore it to build a map that it can use for getting from A to B.
- Methods for escaping from labyrinths – required knowledge for aspiring heroes of antiquity – are also examples of online search algorithms.
- Spatial exploration is not the only form of exploration, however.
- Consider a new born baby : it has many possible actions, but knows the outcomes of none of them, and it has experienced only a few of the possible states that it can reach.
- The baby's gradual discovery of how the world works is, in part, an online search process.

Online search problems

- An online search problem can be solved only by an agent executing actions, rather than by a purely computational process.

- We will assume that the agent knows just the following:
    a. ACTIONS(S), which returns a list of actions allowed in state S.
    b. The step cost function c (s,a,ś) – note that this cannot be used until the agent knows that ś is the outcome, and
    c. GOAL_TEST(S)

- The agent cannot access the successors of a state except by actually trying all the actions in that state.

- For example, in the maze problem, the agent does not know that going up from (1,1) leads to (1,2); nor, having done that, does it know that going Down will take it back to (1,1).

- This degree of ignorance can be reduced in some applications – for example, a robot explorer might know how its movement actions work and be ignorant only of the locations of obstacles.

A sample Maze problem

3			G
2			
1	S		
	1	2	3

- we will assume that the agent can always recognize a state that it has visited before, and we will assume that the actions are deterministic.

- Finally, the agent might have access to an admissible heuristic function h(s) that estimates the distance from the current state to a goal state.

- For example, in maze problem, the agent might know the location of the goal and be able to use the Manhattan distance heuristic.

- Typically, the agent's objective is to reach a goal state while minimizing cost. The cost is the total path cost of the path that the agent actually travels.

- It is common to compare this cost with the path cost of the path the agent would follow if it know the search space in advance – that is, the actual shortest path

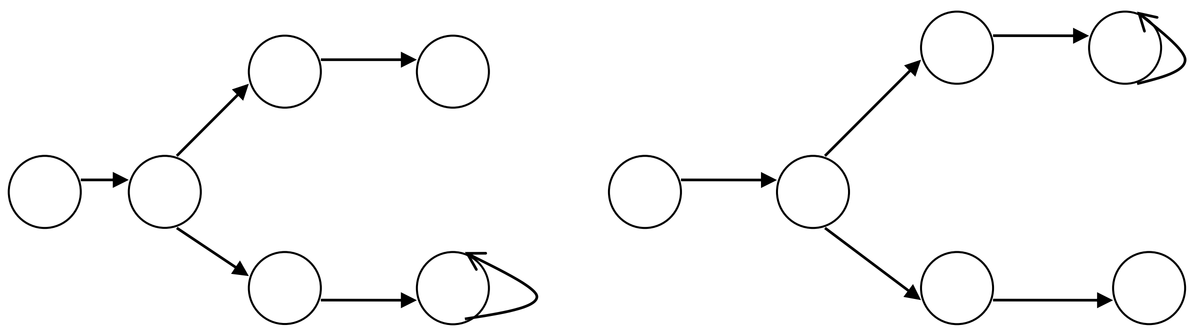
- In the language of online algorithms, this is called the competitive ratio; we would like it to be as small as possible.

- Although this sounds like a reasonable request, it is easy to see that the best achievable competitive ratio is infinite in some cases.

- For example, if some actions are irreversible, the online search might accidentally reach a dead-end state from which no goal state is reachable.

- No algorithm can avoid dead ends in all state spaces.

Figure : Two state spaces that might lead an online search agent into a dead end

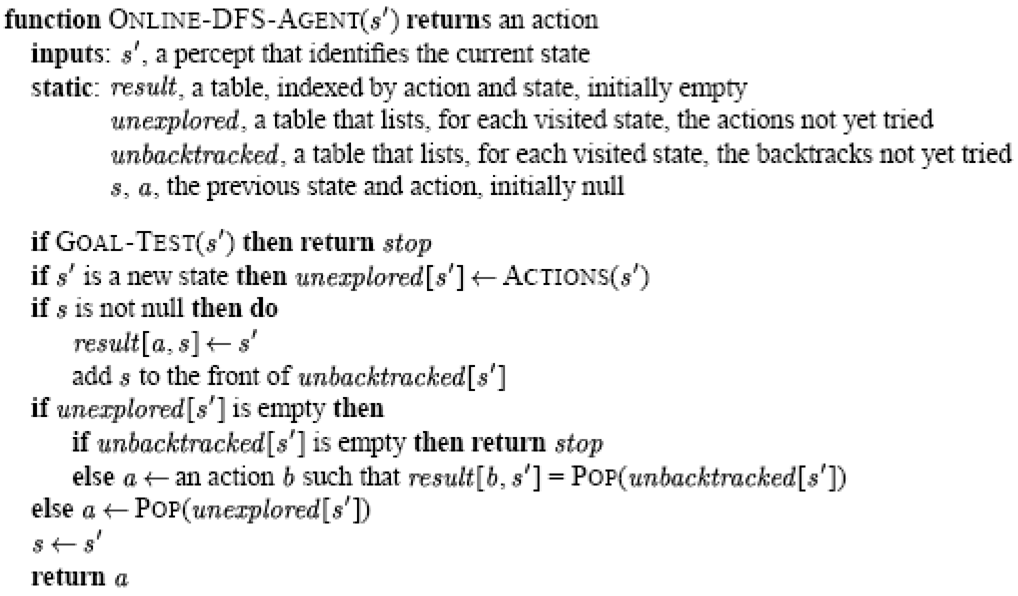


- To an online search algorithm that has visited state S and A, the two state spaces look identical, so it must make the same decision in both. Therefore it will fail in one of them – this is an example of an adversary argument – we can imagine an adversary that constructs the state space while the agent explores it and can put the goals and dead ends wherever it likes

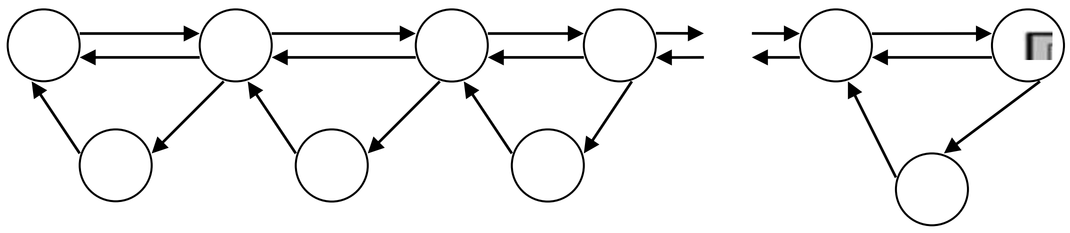
- Dead ends are a real difficulty for robot exploration – staircases, ramps, cliffs and all kinds of natural terrain present opportunities for irreversible actions.

- To make progress, we will simply assume that the state space is safely **explorable** – that is, some goal state is reachable from every reachable state.

State spaces with reversible actions, such as mazes and 8-puzzles, can be viewed as undirected graphs and are clearly safely explorable.



```
function ONLINE-DFS-AGENT(s') returns an action
   inputs: s', a percept that identifies the current state
   static: result, a table, indexed by action and state, initially empty
           unexplored, a table that lists, for each visited state, the actions not yet tried
           unbacktracked, a table that lists, for each visited state, the backtracks not yet tried
           s, a, the previous state and action, initially null

   if GOAL-TEST(s') then return stop
   if s' is a new state then unexplored[s'] ← ACTIONS(s')
   if s is not null then do
       result[a, s] ← s'
       add s to the front of unbacktracked[s']
   if unexplored[s'] is empty then
       if unbacktracked[s'] is empty then return stop
       else a ← an action b such that result[b, s'] = POP(unbacktracked[s'])
   else a ← POP(unexplored[s'])
   s ← s'
   return a
```



- The basic idea is to store a "current best estimate" H(s) of the cost to reach the goal from each state that has been visited.
- H(s) starts out being just the heuristic estimate h(s) and is updated as the agent gains experience in the state space.

- In (a), the agent seems to be stuck in a flat local maximum at the shaded state.

- Rather than staying where it is, the agent should follow what seems to be the best path to the goal based on the current cost estimates for its neighbors.

The estimated cost to reach the goal through a neighbor ś plus the estimated cost to get to a goal from there – that is, C(S,a, ś) +H(ś)

8. EXAMPLE : REAL WORLD PROBLEMS

 A real world problem is one whose solutions people actually care about.

 They tend not to have a single agreed upon description, but attempt is made to give general flavor of their
formulation,

 The following are the some real world problems,

o Route Finding Problem

o Touring Problems

o Travelling Salesman Problem

o Robot Navigation

ROUTE-FINDING PROBLEM

 Route-finding problem is defined in terms of specified locations and transitions along links between
them.

 Route-finding algorithms are used in a variety of applications, such as routing in computer networks, military operations planning, and air line travel planning systems.

AIRLINE TRAVEL PROBLEM

The **airline travel problem** is specifies as follows **:**

o **States :** Each is represented by a location(e.g.,an airport) and the current time.
o **Initial state :** This is specified by the problem.
o **Successor function :** This returns the states resulting from taking any scheduled flight(further specified by seat class and location),leaving later than the current time plus the within-airport transit time,from the current airport to another.
o **Goal Test :** Are we at the destination by some prespecified time?
o **Path cost :** This depends upon the monetary cost,waiting time, flight time, customs and immigration procedures,seat quality,time of dat,type of air plane,frequent-flyer mileage awards, and so on.

TOURING PROBLEMS

 Touring problems are closely related to route-finding problems,but with an important difference.

 Consider for example, the problem, "Visit every city at least once" as shown in
Romania map.

 As with route-finding the actions correspond to trips between adjacent cities. The state space, however,is quite different.

Initial state would be "In Bucharest; visited{Bucharest}".
Intermediate state would be "In Vaslui; visited {Bucharest,Vrziceni,Vaslui}".
Goal test would check whether the agent is in Bucharest and all 20 cities have been visited.

THE TRAVELLING SALESPERSON PROBLEM (TSP)

✓ TSP is a touring problem in which each city must be visited exactly once.
✓ The aim is to find the shortest tour. The problem is known to be **NP-hard**.
✓ Enormous efforts have been expended to improve the capabilities of TSP algorithms.
✓ These algorithms are also used in tasks such as planning movements of **automatic circuit-board drills** and of **stocking machines** on shop floors.

VLSI layout

A **VLSI layout** problem requires positioning millions of components and connections on a chip to minimize area ,minimize circuit delays,minimize stray capacitances,and maximize manufacturing yield. The layout problem is split into two parts : **cell layout** and **channel routing**.

ROBOT navigation

ROBOT navigation is a generalization of the route-finding problem. Rather than a discrete set of routes,a robot can move in a continuous space with an infinite set of possible actions and states. For a circular Robot moving on a flat surface,the space is essentially two-dimensional.

When the robot has arms and legs or wheels that also must be controlled,the search space becomes multi-dimensional. Advanced techniques are required to make the search space finite.

AUTOMATIC ASSEMBLY SEQUENCING

The example includes assembly of intricate objects such as electric motors. The aim in assembly problems is to find the order in which to assemble the parts of some objects. If the wrong order is choosen, there will be no way to add some part later without undoing somework already done.
Another important assembly problem is protein design, in which the goal is to find a sequence of Amino acids that will be fold into a three-dimensional protein with the right properties to cure some disease.

INTERNET SEARCHING

In recent years there has been increased demand for software robots that perform Internet searching, looking for answers to questions, for related information, or for shopping deals.
The searching techniques consider internet as a graph of nodes (pages) connected by links.

9. SEARCHING WITH PARTIAL OBSERVATIONS

Beyond Classical Search

- Searching with Nondeterministic Actions
- Searching with Partial Observation
- Online Search Agents and Unknown Environments

When the environment is fully observable and deterministic, and the agent knows what the effects of each action are, percepts provide no new information after the agent determines the initial state.

In partially observable environments, every percept helps narrow down the set of possible states the agent might be in, making it easier for the agent to achieve its goals.

- In nondeterministic environments, percepts tell the agent which of the possible outcomes has actually occurred.
- ・In both cases, future percepts cannot be determined in advance and the agent's future actions will depend on those future percepts.
- ・A solution to this type of problem is a **contingency plan (also know as a strategy)** that specifies what to do depending on what percepts are received.

If the environment is not fully observable or deterministic, then the following types of problems occur:
1. Sensorless problems
If the agent has no sensors, then the agent cannot know it's current state, and hence would have to make many repeated action paths to ensure that the goal state is reached regardless of it's initial state.

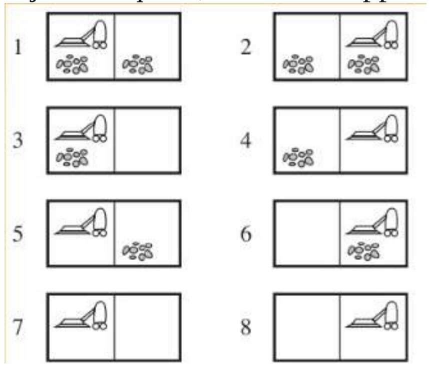
2. Contingency problems
This is when the environment is partially observable or when actions are uncertain. Then after each action the agent needs to verify what effects that action has caused. Rather than planning for every possible contingency after an action, it is usually better to start acting and see which contingencies do arise.
This is called interleaving of search and execution.

A problem is called adversarial if the uncertainty is caused by the actions of another agent.

Searching with Nondeterministic Actions

Example: Erratic Vacuum World, same state space as before. Goal states are 7 and 8.

Suck action is:
When applied to a dirty square, the action cleans the square and sometimes cleans up dirt in an adjacent square, too When applied to a clean square, the action sometimes deposits dirt on the carpet.



To formulate this problem, generalize notion of transition model from before. Use Results function that returns a ***set*** of possible outcome states.
E.g., Results (1, *Suck*) = { 5, 7 }

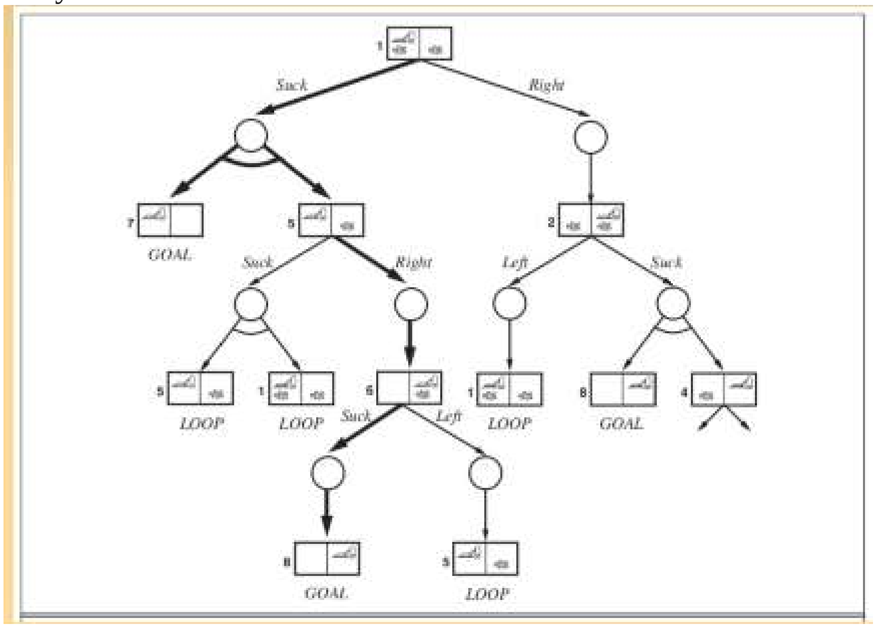
Also generalize notion of a solution to a contingency plan.
E.g., From state 1, [*Suck*, if *State* = 5 then [*Right*, *Suck*] else [ ]]

Augment search trees in the following way

Branching caused by agent's choice of action are called *OR nodes*. E.g., in vacuum world, agent chooses *Left* or *Right* or *Suck*.

Branching caused by environment's choice of action are called *AND nodes*. E.g., in erratic vacuum world, *Suck* action in state 1 leads to a state in { 5, 7 }, so agent would need to find a plan for state 5 and state 7.

Two kinds of nodes alternate giving *AND-OR tree*.
Solution is a subtree that has a goal node at every leaf specifies one action at each -OR node includes every outcome branch at each AND node.



```
Function: AND-OR-Graph-Search
Receives: *problem*; Returns: *conditional plan* or *failure*
**OR-Search (*problem*.InitialState, *problem*, [ ])**
**Function: OR-Search**
Receives: *state*, *problem*, *path*
Returns: *conditional plan* or *failure*
        1. If *problem*.GoalTest(*state*) then return empty plan
        2. If *state* is on path then return failure
        3. For each action in *problem*.Actions(*state*) do
           3.1 *plan* = AND-Search (Results(*state*, *action*),
                                   *problem*, [ *state* | *path* ])
           3.2 if *plan* □ *failure* then return [ *action* | *plan* ]
        4. Return *failure*

**Function: AND-Search**
Receives: *states*, *problem*, *path*
Returns: *conditional plan* or *failure*
1. For each *s*  in *states* do
   1.1 *plan ii* = OR-Search(*s* , *problem*, *path*)
   1.2 If *plan ii*  = *failure* then return *failure*
2. Return [ if *s1* then *plan1* else if *s2* then *plan* else …
   if *sn-1* then *plan n-1* else *plan n* ] *2*
```

Note loops are handled by looking for a state in the current path and returning failure. This guarantees termination in finite state spaces.

Given algorithm is DFS. Can also search tree with BFS or best-first, and generalize heuristic function for contingency plans for an analog to A*.

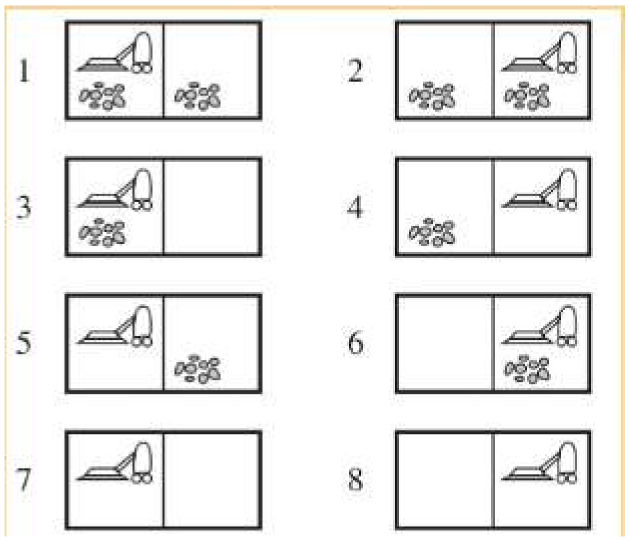
An alternative agent design is for the agent act ***before*** it has a guaranteed plan and deal contingencies only as they arise in execution. This type of ***interleaving*** of search and execution is useful for exploration problems and game playing.

As noted, when an environment is partially observable, an agent can be in one of several possible states. An action leads to one of several possible outcomes.

To solve these problems, an agent maintains a ***belief state*** that represent the agent's current belief about the possible physical state it might be in, given the sequence of actions and percepts up to that point.

When an agent's percepts provide no information at all, this is called a ***sensorless*** (or ***conformant***) problem Example: deterministic, assume know geography, but not location or dirt.

Initially could be in any state { 1, 2, 3, 4, 5, 6, 7, 8 }
Do *Right*, must in { 2, 4, 6, 8 }.
Do *Suck* results in { 4, 8 }.
Doing *Left*, then *Suck*
coerces world into state 7 regardless of initial state



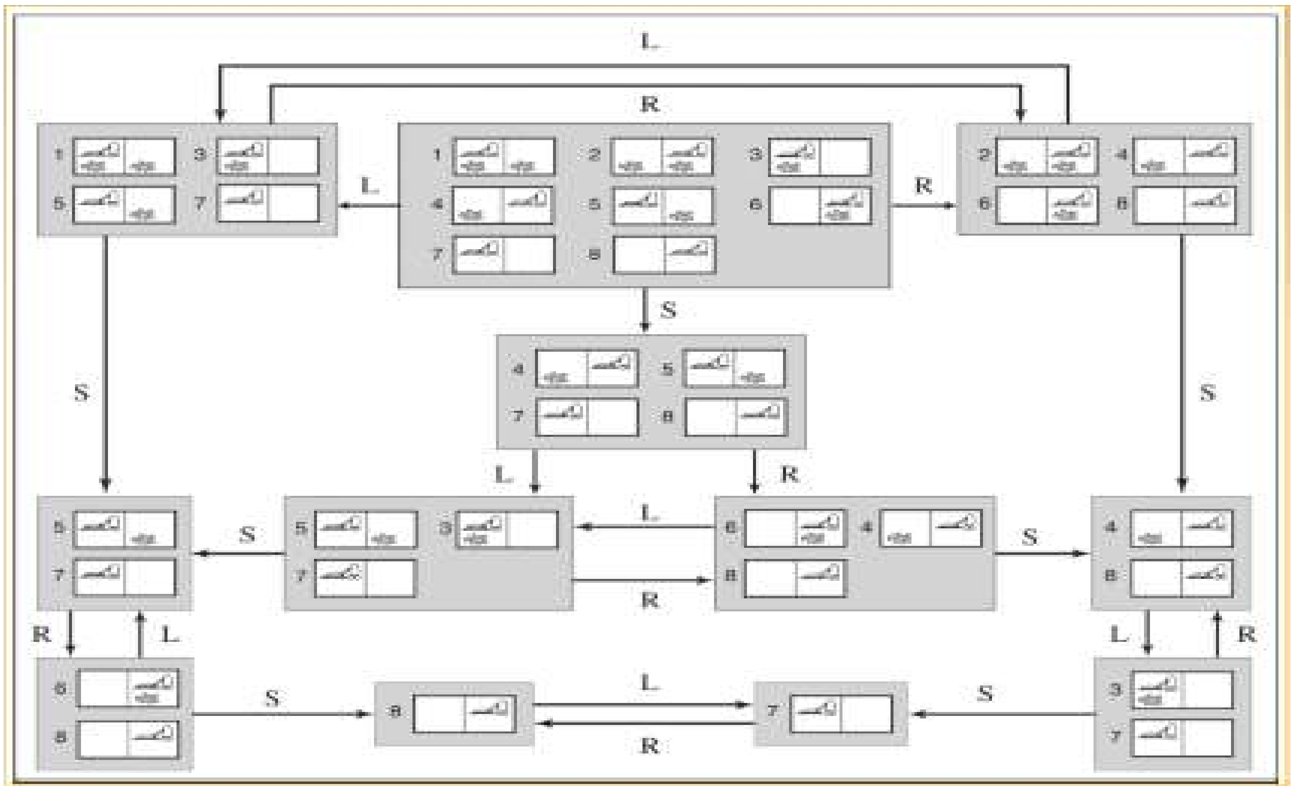
Solve sensorless problems by searching space of belief states, which is fully observable to agent. Suppose physical problem *P* with Actions *P* , Result *P* , GoalTest *P* and StepCost
Define corresponding sensorless problem:

Belief states: powerset of states of *P*, although many are unreachable from initial state. For *N* states of *P*, there are 2 *N* possible belief states.

Initial state: typically, all the states of *P*
Actions: if illegal actions are safe, use the union of actions of all states in *b*; otherwise use the intersection. Use to ***predict*** the next belief states.
Transition model: *b'* = Result (*b*, *a*) = { *s'* : *s'* = Result (*s*, *a*) and *s* □ *b* }.
Similarly for nondeterministic environments using Results *P*

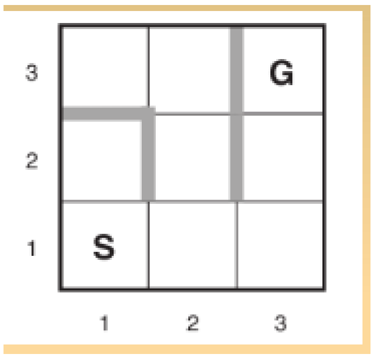
Goal test: ***all*** of the physical states in *b* must satisfy GoalTest *P*
Path cost: Assume that action costs same in all states, so transfer from underlying problem

Complete belief state search space for deterministic environment.
Note only 12 reachable states out of 2 possible belief states.



Previous problems use ***offline search*** algorithms. Complete solution is computed, then solution is executed.

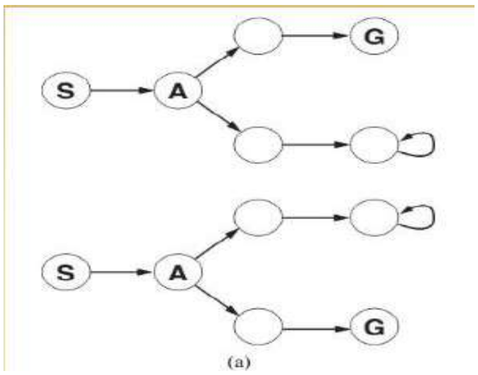
In ***online search***, agent interleaves computation with action: it first takes an action, then observes the environment and computes the next action.
Good idea for dynamic environments where there is a penalty for computing too long. Helpful idea for nondeterministic environments. Don't spend time planning for contingencies that are rare.
Necessary idea for unknown environments, where agent doesn't know what states exists or the results of its actions. Called an ***exploration problem***. Agent uses actions as experiments to learn about its environment
Assume deterministic, fully observable environment. Agent only knows:
Actions(*s*) – list of actions allowed in state s Step-cost function *c*(*s*, *a*, *s'*) – cannot be used until agent knows that *s'* is the outcome of doing *a* GoalTest(*s*) In particular, it doesn't know Result (*s*,*a*) except by actually being in *s* and doing *a*, then observing *s'*.



Example: maze problem.
Agent starts in S. Goal is to move to G. Agent initially does not know that going *Up* from (1,1) leads to (1,2) nor that going *Down* from there goes back to (1,1).
Agent may have access to admissible heuristic.
E.g., if agent knows where goal is, can use Manhattan distance heuristic.

Typically, objective is to reach goal state while minimizing cost, where cost is total path cost of path an agent actually travels. Common to compare this cost with path cost of path agent would follow if it knew the search space in advance, i.e., the actual shortest path. Called the ***competitive ratio***, and would like it to be as small as possible.

Nice idea, but in some cases competitive ratio is infinite, if online search agent reaches a dead-end state from which no goal state is reachable.
Claim: no algorithm can avoid dead ends in all state spaces. Two states shown are indistinguishable, So must result in same action sequence.
Will be wrong for one.



To make progress, we assume that state space is ***safely explorable***. I.e., some goal state is reachable from every reachable state. State spaces with reversible actions (mazes, 8puzzles) are clearly safely explorable.
Even with this assumption, no bounded competitive ratio can be guaranteed if there are paths of unbounded cost. Common to describe performance in terms of size of entire state space instead of depth of the shallowest node Online search algorithms are very different from offline search algorithms. Since agent occupies specific physical node, can only expand immediate successors. Need to expand nodes in ***local*** order. DFS has this property, but needs reversible actions to support backtracking.

Hill-climbing already is a local search algorithm, but can get stuck in local maxima. Add memory to keep track of a "current best estimate", $H(s)$, of the cost to reach goal from each state that has been visited.
$H(s)$ starts out with $h(s)$, the heuristic estimate and is updated as agent gains experience

10. CONSTRAINT SATISFACTION PROBLEMS

A Constraint Satisfaction Problem is characterized by:

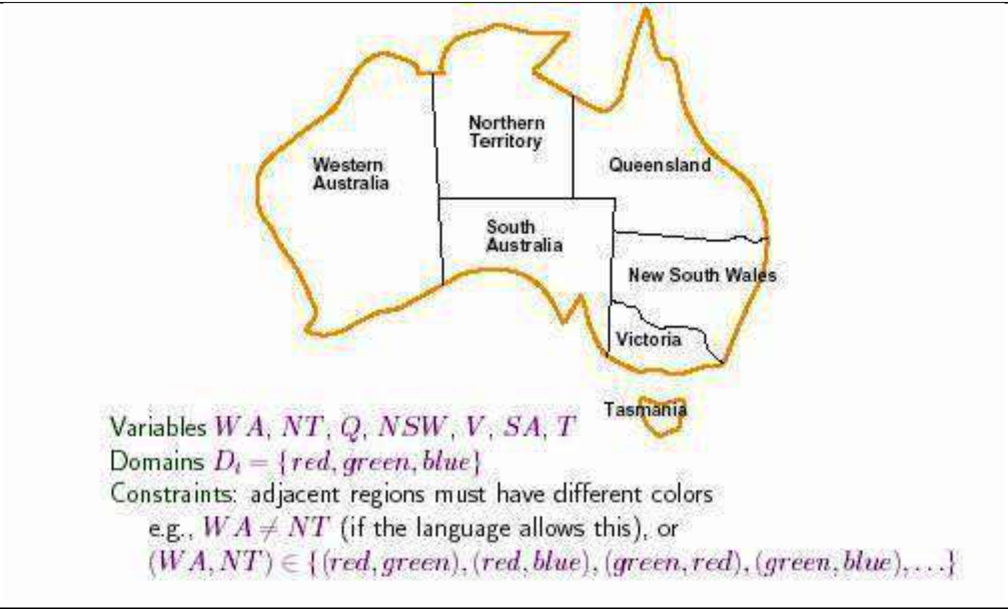
- a *set of variables* {x1, x2, .., xn},
- for each variable xi a *domain* Di with the possible values for that variable, and
- a set of *constraints*, i.e. relations, that are assumed to hold between the values of the variables. [These relations can be given intentionally, i.e. as a formula, or extensionally, i.e. as a set, or procedurally, i.e. with an appropriate generating or recognising function.] We will only consider constraints involving one or two variables.

The constraint satisfaction problem is to find, for each i from 1 to n, a value in Di for xi so that all constraints are satisfied. A CS problem is usually represented as an undirected graph, called *Constraint Graph* where the nodes are the variables and the edges are the binary constraints. Unary constraints can be disposed of by just redefining the domains to contain only the values that satisfy all the unary constraints. Higher order constraints are represented by hyperarcs. In the following we restrict our attention to the case of unary and binary constraints.

Consistency Based Algorithms use information from the constraints to reduce the search space as early in the search as it is possible

- This problem requires a lot of reasoning.
- Time complexity of the problem is more as concerned to the other problems.
- This problem can also be solved by the evolutionary approach and mutation operations.
- This problem is dependent upon some constraints which are necessary part of the problem.
- Various complex problems can also be solved by this technique.

Example



Variables WA, NT, Q, NSW, V, SA, T
Domains $D_i = \{red, green, blue\}$
Constraints: adjacent regions must have different colors
e.g., $WA \neq NT$ (if the language allows this), or
$(WA, NT) \in \{(red, green), (red, blue), (green, red), (green, blue), \ldots\}$

Figure 2.15 (a) Principle states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem. The goal is to assign colors to each region so that no neighboring regions have the same color.

Constraint graph: nodes are variables, arcs show constraints

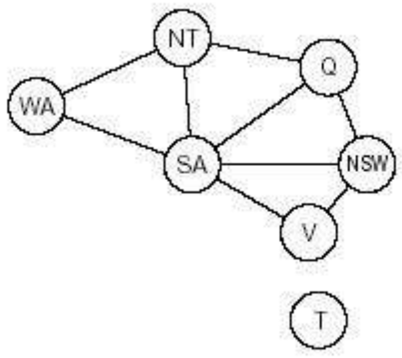
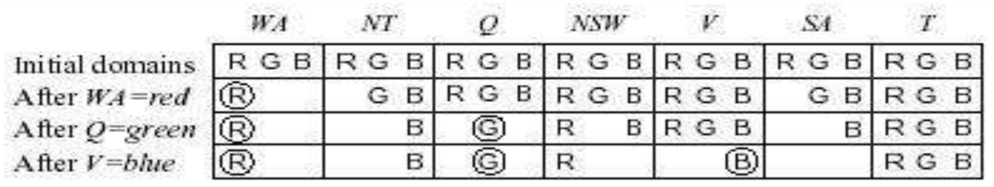


Figure 2.15 (b) The map coloring problem represented as a constraint graph.



	WA	*NT*	*Q*	*NSW*	*V*	*SA*	*T*
Initial domains	R G B	R G B	R G B	R G B	R G B	R G B	R G B
After *WA=red*	Ⓡ	G B	R G B	R G B	R G B	G B	R G B
After *Q=green*	Ⓡ	B	Ⓖ	R B	R G B	B	R G B
After *V=blue*	Ⓡ	B	Ⓖ	R	Ⓑ		R G B

Figure 5.6 The progress of a map-coloring search with forward checking. *WA* = *red* is assigned first; then forward checking deletes *red* from the domains of the neighboring variables *NT* and *SA*. After *Q* = *green*, *green* is deleted from the domains of *NT*, *SA*, and *NSW*. After *V* = *blue*, *blue* is deleted from the domains of *NSW* and *SA*, leaving *SA* with no legal values.

- **CSP can be viewed as a standard search problem as follows :**

- Initial state : the empty assignment {},in which all variables are unassigned.
- Successor function : a value can be assigned to any unassigned variable,provided that it does not conflict with previously assigned variables.
- Goal test : the current assignment is complete.
- Path cost : a constant cost(E.g.,1) for every step.

- Every solution must be a complete assignment and therefore appears at depth n if there are n variables.
- Depth first search algorithms are popular for CSPs

Varieties of constraints :
(i) Unary constraints involve a single variable.
Example: SA # green
(ii) Binary constraints involve pairs of variables.
Example: SA # WA
(iii) Higher order constraints involve 3 or more variables.
Example: cryptarithmetic puzzles.
(iv) Absolute constraints are the constraints, which rules out a potential solution when they are violated
(v) Preference constraints are the constraints indicating which solutions are preferred

11. CONSTRAINT PROPAGATION:INFERENCE IN CSPS
In regular state-space search, an algorithm can do only one thing: search.
In CSPs there is a choice: an algorithm can search (choose a new variable assignment from several possibilities) or do a specific type of **inference** called **constraint propagation**: using the constraints to reduce the number of legal values for a variable, which in turn can reduce the legal values for another variable, and so on.

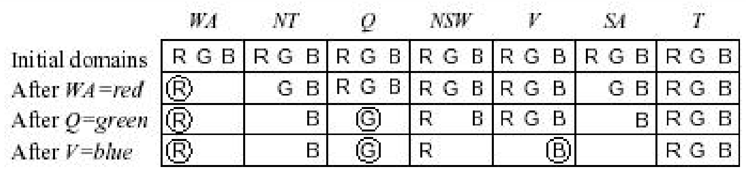
Constraint propagation may be intertwined with search, or it may be done as a preprocessing step, before search starts. Sometimes this preprocessing can solve the whole problem, so no search is required at all.
The key idea is **local consistency**. If we treat each variable as a node in a and each binary constraint as an arc, then the process of enforcing local consistency in each part of the graph causes inconsistent values to be eliminated throughout the graph. There are different types of local consistency, which we now cover in turn.

So far our search algorithm considers the constraints on a variable only at the time that the Variable is chosen by SELECT-VNASSIGNED-VARIABLE.

But by looking at some of the constraints earlier in the search, or even before the search has started, we can drastically reduce the search space.

Forward checking

- One way to make better use of constraints during search is called **forward checking**.
- Whenever a variable X is assigned, the forward checking process looks at each unas signed variable Y that is connected to X by a constraint and deletes from Y 's domain any value that is inconsistent with the value chosen for X.
- The following figure shows the progress of a map-coloring search with forward checking.



	WA	*NT*	*Q*	*NSW*	*V*	*SA*	*T*
Initial domains	R G B	R G B	R G B	R G B	R G B	R G B	R G B
After *WA=red*	Ⓡ	G B	R G B	R G B	R G B	G B	R G B
After *Q=green*	Ⓡ	B	Ⓖ	R B	R G B	B	R G B
After *V=blue*	Ⓡ	B	Ⓖ	R	Ⓑ		R G B

Figure 5.6 The progress of a map-coloring search with forward checking. $WA = red$ is assigned first; then forward checking deletes red from the domains of the neighboring variables NT and SA. After $Q = green$, $green$ is deleted from the domains of NT, SA, and NSW. After $V = blue$, $blue$ is deleted from the domains of NSW and SA, leaving SA with no legal values.

Although forward checking detects many inconsistencies, it does not detect all of them.

- **Constraint propagation** is the general term for propagating the implications of a constraint on one variable onto other variables.

Node consistency

A single variable (corresponding to a node in the CSP network) is **node-consistent** if all the values in the variable's domain satisfy the variable's unary constraints. For example, in the variant of the Australia map-coloring problem where South Australians dislike green, the variable SA starts with domain {red , green, blue}, and we can make it node consistent by eliminating green, leaving SA with the reduced domain {red , blue}.We say that a network is node-consistent if every variable in the network is node-consistent.

Arc Consistency

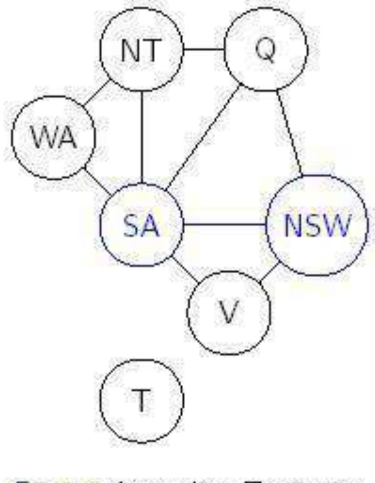
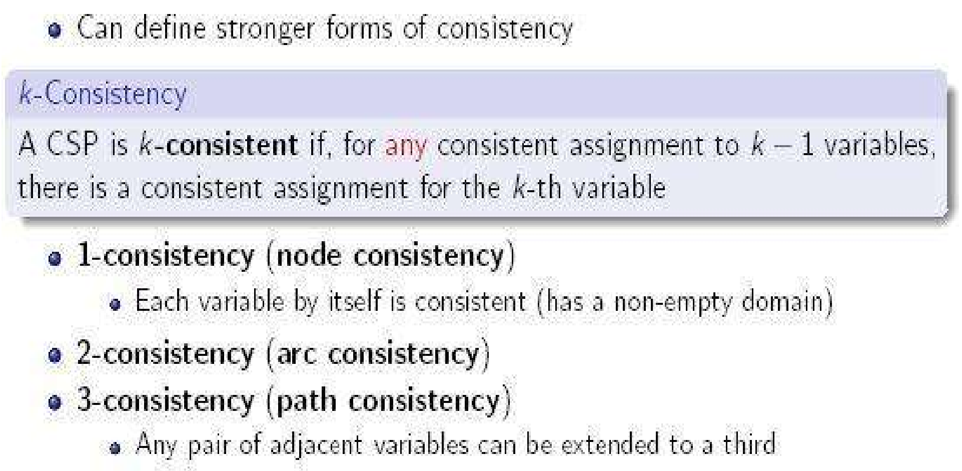


Figure: Australian Territories

- One method of constraint propagation is to enforce **arc consistency**
  - Stronger than forward checking
  - Fast
- *Arc* refers to a *directed* arc in the constraint graph
- Consider two nodes in the constraint graph (e.g., *SA* and *NSW*)
  - An arc is **consistent** if
  - For every value *x* of *SA*
  - There is some value *y* of *NSW* that is consistent with *x*
- Examine arcs for consistency in *both* directions

K-Consistency



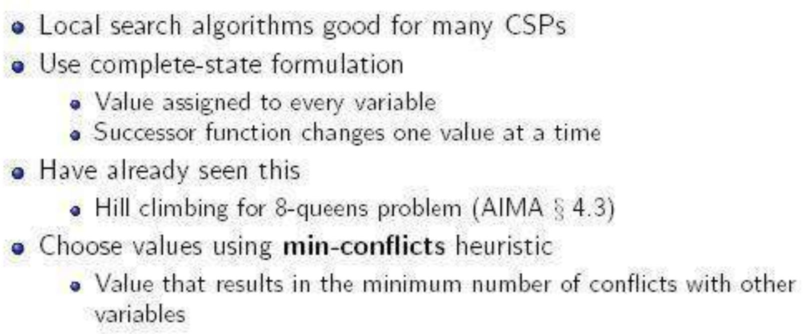
- Can define stronger forms of consistency

k-Consistency

A CSP is ***k*-consistent** if, for any consistent assignment to $k-1$ variables, there is a consistent assignment for the *k*-th variable

- **1-consistency (node consistency)**
  - Each variable by itself is consistent (has a non-empty domain)
- **2-consistency (arc consistency)**
- **3-consistency (path consistency)**
  - Any pair of adjacent variables can be extended to a third

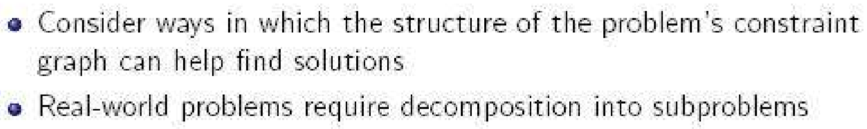
Local Search for CSPs



- Local search algorithms good for many CSPs
- Use complete-state formulation
  - Value assigned to every variable
  - Successor function changes one value at a time
- Have already seen this
  - Hill climbing for 8-queens problem (AIMA § 4.3)
- Choose values using **min-conflicts** heuristic
  - Value that results in the minimum number of conflicts with other variables

The key idea of CSP constraint propagation is **local consistency**. If we treat each variable as a node in a graph and each binary constraint as an arc, then the process of enforcing local consistency in each part of the graph causes inconsistent values to be eliminated throughout the graph. There are different types of local consistency, which we now cover in turn.

The Structure of Problems
Problem Structure



- Consider ways in which the structure of the problem's constraint graph can help find solutions
- Real-world problems require decomposition into subproblems

Independent Sub problems

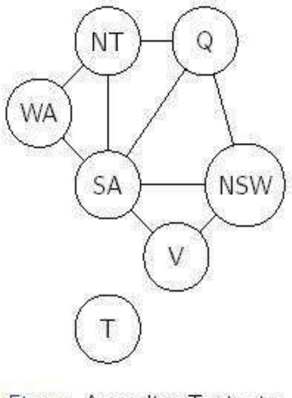
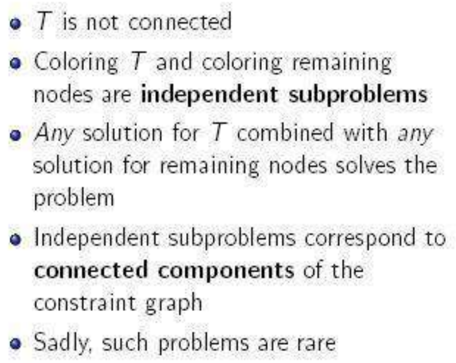


Figure: Australian Territories



- *T* is not connected
- Coloring *T* and coloring remaining nodes are **independent subproblems**
- *Any* solution for *T* combined with *any* solution for remaining nodes solves the problem
- Independent subproblems correspond to **connected components** of the constraint graph
- Sadly, such problems are rare

Tree-Structured CSPs

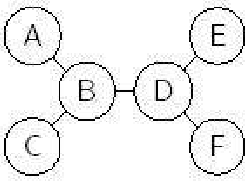


Figure: Tree-Structured CSP

- In most cases, CSPs are connected
- A simple case is when the constraint graph is a **tree**
- Can be solved in time linear in the number of variables
  - Order variables so that each parent precedes its children
  - Working "backward," apply arc consistency between child and parent
  - Working "forward," assign values consistent with parent

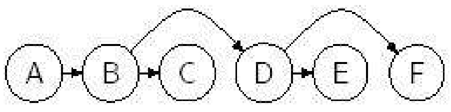


Figure: Linear ordering

The key idea is **local consistency**. If we treat each variable as a node in a graph and each binary constraint as an arc, then the process of enforcing local consistency in each part of the graph causes inconsistent values to be eliminated throughout the graph. There are different types of local consistency, which we now cover in turn.

Global constraints

Global constraint is one involving an arbitrary number of variables (but not necessarily all variables). Global constraints occur frequently in real problems and can be handled by special-purpose algorithms that are more efficient than the general-purpose methods described so far. For example, the Alldiff constraint says that all the variables involved must have distinct values (as in the cryptarithmetic problem above and Sudoku puzzles below). One simple form of inconsistency detection for Alldiff constraints works as follows:
if m variables are involved in the constraint, and if they have n possible distinct values altogether, and m>n, then the constraint cannot be satisfied.

This leads to the following simple algorithm:
First, remove any variable in the constraint that has a singleton domain, and delete that variable’s value from the domains of the remaining variables.
Repeat as long as there are singleton variables. If at any point an empty domain is produced or there are more variables than domain values left, then an inconsistency has been detected.

Resource constraint

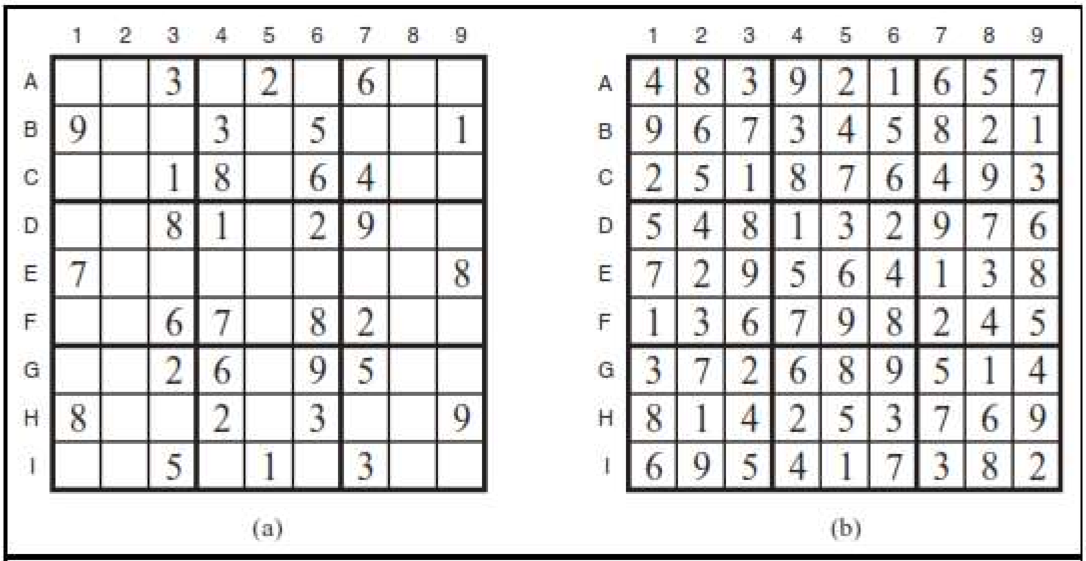
Another important higher-order constraint is the **resource constraint**, sometimes called the atmost constraint. For example, in a scheduling problem, let P denote the numbers of personnel assigned to each of four tasks. The constraint that no more than 10 personnel are assigned in total is written as Atmost (10,P1,P2,P3,P14,...,P).

Instead, domains are represented by upper and lower bounds and are managed by **bounds propagation**. For example, in an airline-scheduling problem, let’s suppose there are two flights, F 1and F2, for which the planes have capacities 165 and 385, respectively. The initial domains for the numbers of passengers on each flight are then D1=[0, 165] and D2=[0, 385] .

Sudoku example

The popular **Sudoku** puzzle has introduced millions of people to constraint satisfaction problems, although they may not recognize it. A Sudoku board consists of 81 squares, some of which are initially filled with digits from 1 to 9. The puzzle is to fill in all the remaining squares such that no digit appears twice in any row, column, or 3 ×3 box (see Figure 6.4). A row, column, or box is called a **unit**.

The Sudoku puzzles that are printed in newspapers and puzzle books have the property that there is exactly one solution. Although some can be tricky to solve by hand, taking tens of minutes, even the hardest Sudoku problems yield to a CSP solver in less than 0.1 second. Sudoku puzzle can be considered a CSP with 81 variables, one for each square. We use the variable names A1 through A9 for the top row (left to right), down to I1 through I9 for the bottom row. The empty squares have the domain {1, 2, 3, 4, 5, 6, 7, 8, 9} and the pre- filled squares have a domain consisting of a single value. In addition, there are 27 different



(a)

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

(b)

	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

Figure 6.4 (a) A Sudoku puzzle and (b) its solution.

Alldiff constraints: one for each row, column, and box of 9 squares.

$$Alldiff(A1, A2, A3, A4, A5, A6, A7, A8, A9)$$
$$Alldiff(B1, B2, B3, B4, B5, B6, B7, B8, B9)$$
$$\cdots$$
$$Alldiff(A1, B1, C1, D1, E1, F1, G1, H1, I1)$$
$$Alldiff(A2, B2, C2, D2, E2, F2, G2, H2, I2)$$
$$\cdots$$
$$Alldiff(A1, A2, A3, B1, B2, B3, C1, C2, C3)$$
$$Alldiff(A4, A5, A6, B4, B5, B6, C4, C5, C6)$$

It is interesting to note how far we can go without saying much that is specific to Sudoku. We do of course have to say that there are 81 variables, that their domains are the digits 1 to 9, and that there are 27 *Alldiff* constraints. But beyond that, all the strategies—arc consistency, path consistency, etc.—apply generally to all CSPs, not just to Sudoku problems. Even naked triples is really a strategy for enforcing consistency of *Alldiff* constraints and has nothing to do with Sudoku *per se*. This is the power of the CSP formalism: for each new problem area, we only need to define the problem in terms of constraints; then the general constraint-solving mechanisms can take over.

12. BACKTRACKING SEARCH

The term **backtracking search** is used for depth-first search that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign.

```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return BACKTRACK({ }, csp)

function BACKTRACK(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
      if value is consistent with assignment then
          add {var = value} to assignment
          inferences ← INFERENCE(csp, var, value)
          if inferences ≠ failure then
             add inferences to assignment
             result ← BACKTRACK(assignment, csp)
             if result ≠ failure then
                return result
      remove {var = value} and inferences from assignment
  return failure
```

Figure 6.5 A simple backtracking algorithm for constraint satisfaction problems. The algorithm is modeled on the recursive depth-first search of Chapter 3. By varying the functions SELECT-UNASSIGNED-VARIABLE and ORDER-DOMAIN-VALUES, we can implement the general-purpose heuristics discussed in the text. The function INFERENCE can optionally be used to impose arc-, path-, or k-consistency, as desired. If a value choice leads to failure (noticed either by INFERENCE or by BACKTRACK), then value assignments (including those made by INFERENCE) are removed from the current assignment and a new value is tried.

The term **backtracking search** is used for a depth-first search that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign. It repeatedly chooses an unassigned variable, and then tries all values in the domain of that variable in turn, trying to find a solution. If an inconsistency is detected, then BACKTRACK returns failure, causing the previous call to try another value. Part of the search tree for the Australia problem is shown in Figure 6.6, where we have assigned variables in the order WA, NT,Q,.... Because the representation of CSPs is tandardized, there is no need to supply BACKTRACKING-SEARCH with a domain-specific initial state, action function, transition model, or goal test.

Notice that BACKTRACKING-SEARCH keeps only a **single representation of a state and alters that representation rather than creating new ones**.

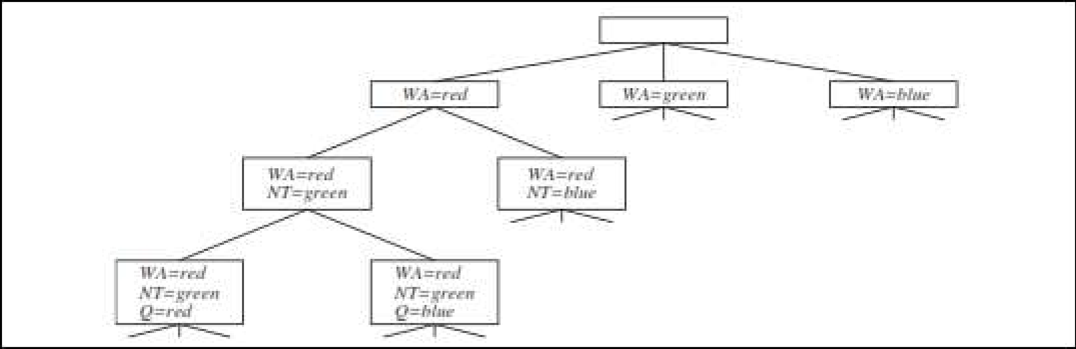


Figure 6.6 Part of the search tree for the map-coloring problem in Figure 6.1.

It turns out that we can solve CSPs efficiently *without* such domain-specific knowledge. Instead, we can add some sophistication to the unspecified functions, using them to address the following questions:

1. Which variable should be assigned next (SELECT-UNASSIGNED-VARIABLE), and in what order should its values be tried (ORDER-DOMAIN-VALUES)?

2. What inferences should be performed at each step in the search (INFERENCE)?
3. When the search arrives at an assignment that violates a constraint, can the search avoid repeating this failure?

13. GAME PLAYING

There were **two reasons that games appeared to be a good domain** in which to explore machine intelligence:

- They provide a structured task in which it is very easy to measure success or failure.
- They did not obviously require large amounts of knowledge. They were thought to be solvable by straightforward search from the starting state to a winning position

The first of these reasons remains valid and accounts for continued interest in the area of game playing by machine.
Unfortunately, the second is not true for any but the simplest games.
For example, consider chess.

- The average branching factor is around 35.
- In an average game, each player might make 50 moves.
- So in order to examine the complete game tree, we would have to examine 35^{100} positions.

Thus it is clear that a program that simply does a straight forward search of the game tree will not be able to select even its first move during the lifetime of its opponent.

Some kinds of heuristic search procedure is necessary to improve the effectiveness of a search-based problem-solving program two things can be done:

1. *Improve the generate procedure so that only good moves (or paths) are generated.*
2. *Improve the test procedure so that the best moves (or paths) will be recognized and explored first.*

As the number of legal moves available increases, it becomes increasingly important to apply heuristics to select only those that have some kind of promise.

The ideal way to use a search procedure to find a solution to a problem is to generate moves through the problem space until a goal state is reached.
In the context of game-playing programs, a goal state is one in which we win. Unfortunately, for interesting games such as chess, it is not usually possible, even with a good Plausible- move generator, to search until a goal state is found.

Static evaluation function

In order to choose the best move, the resulting positions must be compared to discover which is most advantageous. This is done using a **static evaluation function**, which uses whatever information it has to evaluate individual board positions by estimating how likely they are to lead eventually to a win.

A very simple static evaluation function for chess based on piece advantage was proposed by Turing—simply add the values of black's pieces (B), the values of white's pieces (W), and then

compute the quotient W/B.

A more sophisticated approach was that taken in Samuel's checkers program, in which the static evaluation function was a linear combination of several simple functions. Thus the complete evaluation function had the form:

c1 x pieceadvantage + c2 x advancement + c3 x centercontrol

Unfortunately, deciding which moves have contributed to wins and which to losses is not always easy. Suppose we make a very bad move, but then, because the opponent makes a mistake, we ultimately win the game. We would not like to give credit for winning to our mistake.

Two important knowledge-based components of a good game-playing program:

- a good plausible-move generator and
- a good static evaluation function.

They must both incorporate a great deal of knowledge about the particular game being played. But unless these functions are perfect, we also need a search procedure that makes it possible to look ahead as many moves as possible to see what may occur.
For a simple one-person game or puzzle, the A* algorithm can be used. It can be applied to reason forward from the current state as far as possible in the time allowed. The heuristic function h' can be applied at terminal nodes and used to propagate values back up the search graph so that the best next move can be chosen. But because of their adversarial nature, this procedure is inadequate for two-person games such as chess.
As values are passed back up, different assumptions must be made at levels where the program chooses the move and at the alternating levels where the opponent chooses. There are several ways that this can be done. The most commonly used method is the minimax procedure.
An alternative approach is the B* algorithm which works on both standard problem-solving trees and on game trees.

MINIMAX SEARCH PROCEDURE

- The minimax search procedure is a depth-first, depth-limited search procedure.
- The idea is to *start at the current position and use the plausible-move generator* to generate the set of possible successor positions.
- *Static evaluation function* is applied to the positions and simply choose the best one. After doing so, the values are backed up to the starting position to represent the evaluation of it.

The starting position is exactly as good for us as the position generated by the best move we can make next. Here we assume that the static evaluation function returns large values to indicate good situations for us, so our goal is to maximize the value of the static evaluation function of the next board position.
An example of this operation is shown in figure 2.1. It assumes a static evaluation functions that returns values ranging from -10 to 10, with 10 indicating a win for us, -10 a win for the opponent, and 0 an even match. Since our goal is to maximize the value of the heuristic function, we choose to move to B. Backing B's value up to A, we can conclude that A's value is 8, since we know we can move to a position with a value of 8.

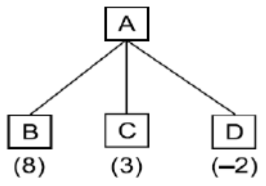


Figure 2.1 One Ply Search

But since we know that the static evaluation function is not completely accurate, we would like to carry the search farther ahead than one ply.
This could be very important, for example, in a chess game in which we are in the middle of a piece exchange. After our move, the situation would appear to be very good, but, if we look one move

ahead, we will see that one of our pieces also gets captured and so the situation is not as favorable as it seemed.

- **<u>Look ahead to see what will happen to each of the new game positions at the next move</u>** which will be made by the opponent.
- Instead of applying the static evaluation function to each of the positions that we just generated, we apply the **<u>plausible-move generator, generating a set of successor positions for each position.</u>**
- If we wanted to stop here, at two-ply look ahead, we could apply the static evaluation function to each of these positions, as shown in figure 2.2.

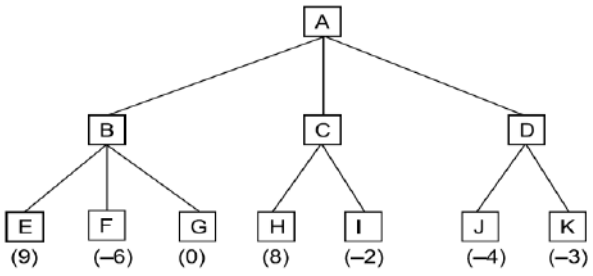


Figure 2.2: Two Ply Search

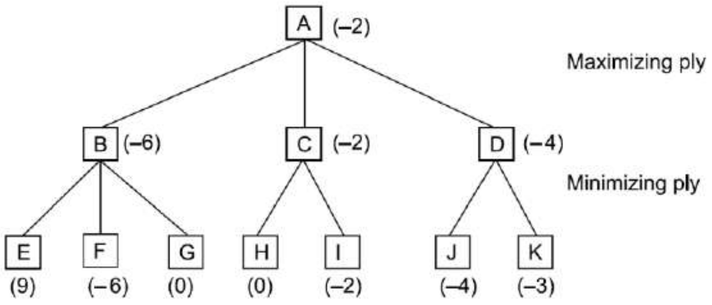


Figure 2.3: **Backing Up the Values of a Two – Ply Search**

But now we must take into account that the opponent gets to choose which successor moves to make and thus which terminal value should be backed up to the next level.
Suppose we made move B. Then the opponent must choose among moves E,F, and G.
The opponent's goal is to minimize the value of the evaluation function, so he or she can be expected to choose move F. This means that if we make move B, the actual position in which we will end up one move later is very bad for us. This is true even though a possible configuration is that represented by node E, which is very good for us. But since at this level we are not the ones to move, we will not get to choose it. Figure 2.3 shows the result of propagating the new values up the tree.

- **At the level representing the opponent's choice, the minimum value was chosen and backed up. At the level representing our choice, the maximum value was chosen**.

Once the values from the second ply are backed up, it becomes clear that the correct move for us to make at the first level, given the information we have available, is C, since there is nothing the opponent can do from there to produce a value worse than -2.
This process can be repeated for as many ply as time allows, and the more accurate evaluations that are produced can be used to choose the correct move at the top level.

- *The alternation of maximizing at alternate ply when evaluations are being pushed back up corresponds to the opposing strategies of the two players and gives this method the name minimax.*

Game can be formally defined as a search problem as below:

- S_0: The **initial state**, which specifies how the game is set up at the start.
- PLAYER(s): Defines which player has the move in a state.
- ACTIONS(s): Returns the set of legal moves in a state.
- RESULT(s, a): The **transition model**, which defines the result of a move.
- TERMINAL-TEST(s): A **terminal test**, which is true when the game is over and false otherwise. States where the game has ended are called **terminal states**.
- UTILITY(s, p): A **utility function** (also called an objective function or payoff function), defines the final numeric value for a game that ends in terminal state s for a player p. In

Minimax algorithm

```
function MINIMAX-DECISION(state) returns an action
    inputs: state, current state in game

    return the a in ACTIONS(state) maximizing MIN-VALUE(RESULT(a, state))
--------------------------------------------------------------------------
function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    $v \leftarrow -\infty$
    for a, s in SUCCESSORS(state) do $v \leftarrow$ MAX(v, MIN-VALUE(s))
    return v
--------------------------------------------------------------------------
function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    $v \leftarrow \infty$
    for a, s in SUCCESSORS(state) do $v \leftarrow$ MIN(v, MAX-VALUE(s))
    return v
```

Types of games:

	deterministic	chance
perfect information	chess, checkers, go, othello	backgammon monopoly
imperfect information	battleships, blind tictactoe	bridge, poker, scrabble nuclear war

It is a straightforward recursive procedure that relies on two auxiliary procedures that are specific to the game being played:

1. MOVEGEN(Position, Player)-The plausible-move generator, which returns a list of nodes representing the move that can be made by Player in Positions.

2. STATIC(Position, Player)- The static evaluation function, which returns a number representing the goodness of position from the standpoint of player^2.

As with any recursive program, a critical issue in the design of the MINIMAX procedure is when to stop the recursion and simply call the static evaluation function. There are a variety of factors that may influence this decision. They include:

- Has one side won?
- How many ply have we already explored?
- How promising is this path?
- How much time is left?
- How stable is the configuration?

DEEP-ENOUGH, which is assumed to evaluate all of these factors and to return TRUE if the search should be stopped at the current level and FALSE otherwise. Our simple implementation of DEEP-ENOUGH will take two parameters, Position and Depth. It will ignore its Position parameter and simply return TRUE if its Depth parameter exceeds a constant cutoff value.

One problem that arises in defining MINIMAX as a recursive procedure is that it needs to return not one but two results:

- The backed-up value of the path it chooses.
- The path itself. We return the entire path even though probably only the first element, representing the best move from the current position, is actually needed.

We assume that MINIMAX returns a structure containing both results and that we have two functions, VALUE and PATH, that extract the separate components.

Since we define the MINIMAX procedure as a recursive functions, we must also specify how it is to be called initially. It takes three parameters, a board position, the current depth of the search, and the player to move. So the initial call to compute the best move from the position CURRENT should be

MINIMAX(CURRENT, 0,PLAYER-ONE)   if PLAYER-ONE is to move, or
MINIMAX(CURRENT,0,PLAYER-TWO)   if PLAYER-TWO is to move.

MINIMAX(Position, Depth, Player)

1. If DEEP-ENOUGH(Position, Depth), then return the structure
   VALUE=STATIC(Position, Player);
   PATH=nil

This indicated that there is no path from this node and that its value is that determined by the static evaluation functions.

2. Otherwise, generate one more ply of the tree by calling the function MOVE-GEN(Position Player) and setting SUCCESSORS to the list it returns.

3. If SUCCESSORS is not empty, then there are no moves to be made, so return the same structure that would have been that would have been returned if DEEP-ENOUGH had returned true.

4. If SUCCESSORS is not empty, then examine each element in turn and keep track of the best one. This is done as follows.

Initialize BEST-SCORE to the minimum value that STATIC can return. It will be updated to reflect the best score that can be achieved by an element of SUCCESSORS.

For each element SUCC of SUCCESSORS, do the following:

(a) Set RESULT-SUCC TO
    MINIMAX(SUCC, Depth+1, OPPOSITE(Player))[3]

This recursive call to MINIMAX will actually carry out the exploration of SUCC.

(b) Set NEW-VALUE to-VALUE(RESULT-SUCC).This will cause it to reflect the merits of the position from the opposite perspective from that of the next lower level.

(c)If NEW-VALUE > BEST-SCORE, Then we have found a successor that is better than any that have been examined so far. Record this by doing the following:

i. Set BEST-SCORE to NEW-VALUE.

ii. The best known path is now from CURRENT to SUCC and then on to the appropriate path down from SUCC as determined by the recursive call to MINIMAX, So set BEST-PATH to the result of attaching SUCC to the front of PATH(RESULT-SUCC).

5. Now that all the successors have been examined, we know the value of Position as well as which path to take form it. So return the structure.

VALUE=BEST-SCORE

PATH=BEST-PATH

When the initial call to MINIMAX returns, the best move from CURRENT is the first element of PATH.

14. OPTIMAL DECISIONS IN GAMES –ALPHA -BETA PRUNING

The MINMAX search procedure is slightly modified to handle both maximizing and minimizing players. It is also necessary to modify the branch and bound strategy to include two bounds, one for each of the players. This modified strategy is called alpha beta pruning.

It requires the maintenance of two threshold values,

- Alpha α : **lower bound on the value that a maximizing node** may ultimately be assigned
- Beta β : **upper bound on the value that a minimizing** node may be assigned.

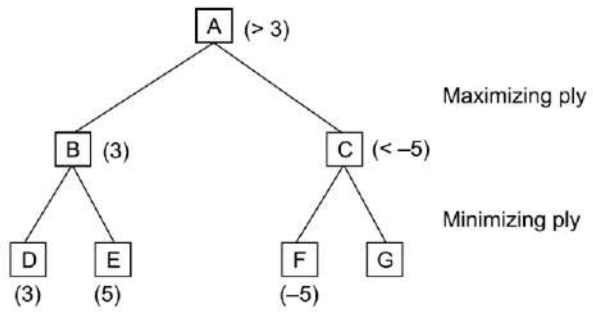


Figure 2.4 An Alpha Cutoff

After examining node F, we know that the opponent is guaranteed a score of -5 or less at C (since the opponent is the minimizing player). But we also know that we are guaranteed a score of 3 or greater at node A which we can achieve if we move to B.

Any other move that produces a score of less than 3 is worse than the move to B, and we can ignore it. After examining only F we are sure that a move to C is worse (it will be <=-5) regardless of the score of node G. <u>Thus we need not bother to explore node G at all.</u> Of course, cutting out one node may not appear to justify the expense of keeping track of the limits and checking them, but if we were exploring this tree to 6 ply, then we would have eliminated not a single node but an entire tree 3 ply deep.

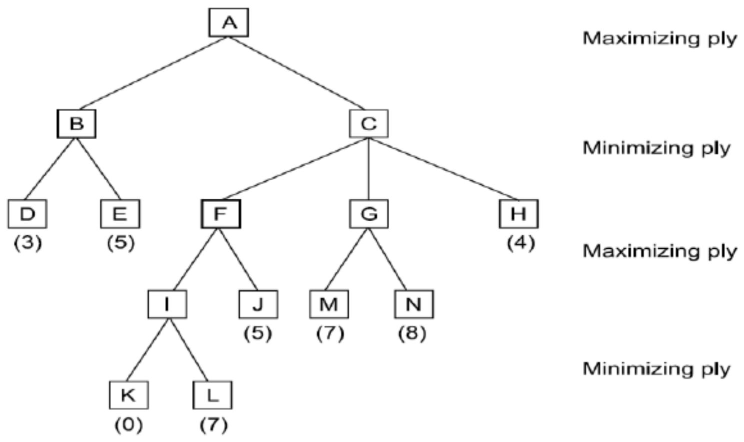


Figure 2.5 Alpha and Beta Cutoffs

To see how the two thresholds, alpha and beta can both be used, consider the example shown in figure 2.5. In searching this tree the entire sub tree headed by B is searched, and we discover that at A we can expect a score of at least 3.

When this alpha value is passed down to F**, it will enable us to skip the exploration of L**. After K is examined, we see that I is guaranteed a maximum score of 0, which means that F is guaranteed a minimum of 0. But this is less than alpha's value of 3, so no more branches of I need to be considered. The maximizing player already knows not to choose to move to C and then to I since, if that move is made, the resulting score will be no better than 0 and a score of 3 can be achieved by moving to B instead.

Now let's see how the value of beta can be used. After cutting off further exploration of I, J is examined, yielding a value of 5, which is assigned as the value of F (since it is the maximum of 5 and 0). This value becomes the value of beta at node C. it indicates that C is guaranteed to get a 5 or less. Now we must expand G. First M is examined and it has a value of 7, which is passed back to G as its tentative value. But now 7 is compared to beta (5). It is greater, and the player whose turn it is at node C is trying to minimize. So this player will not choose G, which would lead to a score of at least 7, since there is an alternative move to F, which will lead to a score of 5. Thus it is not necessary to explore any of the other branches of G.

From this example,

- At **maximizing levels**, we can rule out a move early if it becomes clear that its **value will be less than the current threshold**,
- At **minimizing levels**, search will be terminated if **values that are greater than the current threshold are discovered**.
- ***At maximizing levels, only beta is used to determine whether to cut off the search, and at minimizing levels only alpha is used.***

ALPHA-BETA SEARCH ALGORITHM

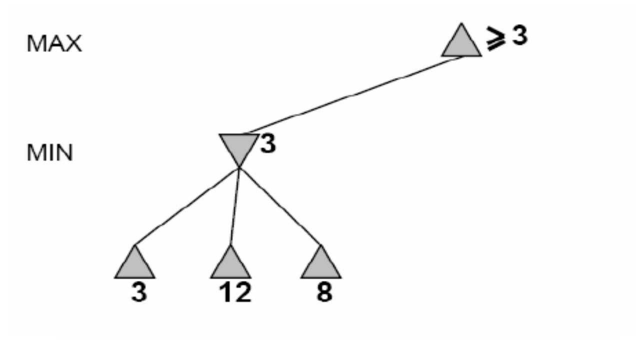
- The problem with minimax search is that the number of game states it has to examine is exponential in the number of moves. Unfortunately we can’t eliminate the exponent, but we can effectively cut it in half. The trick is that it is possible to compute the correct minimax decision without looking at every node in the game tree.

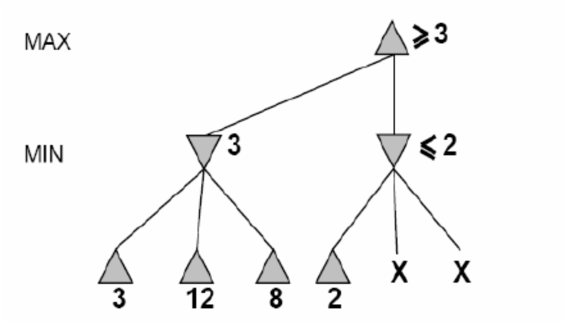
- Alpha-beta pruning technique is applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.

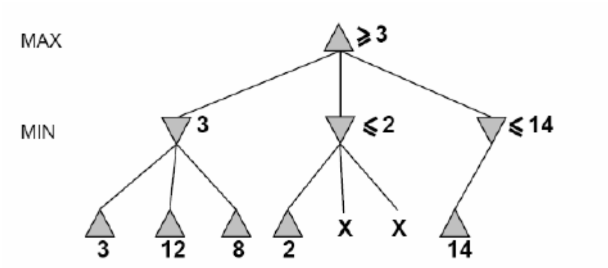
The outcome is that we can identify the minimax decision without ever evaluating two of the leaf nodes.

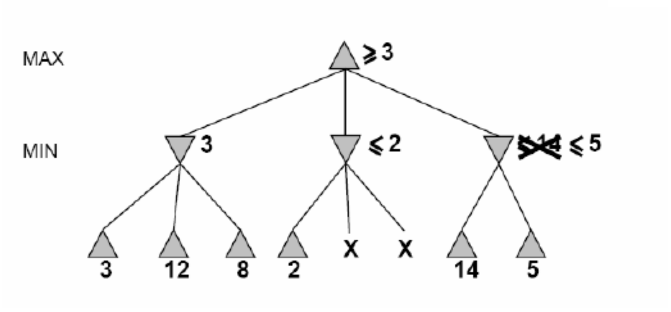
- Another way to look at this is as a simplification of the formula for MINIMAX-VALUE. Let the two unevaluated successors of node C in figure have values x and y and let z be the minimum of x and y. The value of the root node is given by

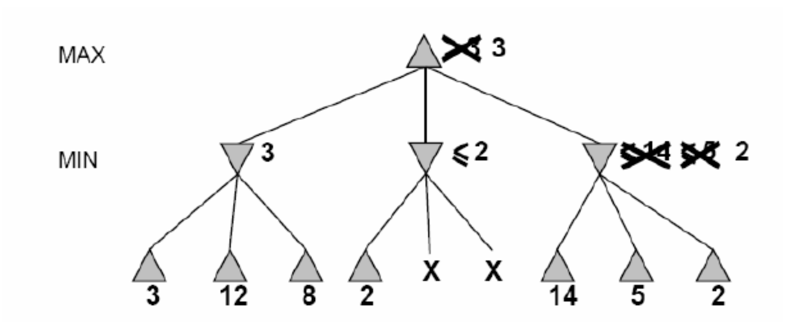
MINIMAX-VALUE(root) = max(min(3,12,8),min(2,x,y), min(14,5,2))
= max (3, min(2,x,y),2)
= max(3,z,2)   where z <= 2
= 3.







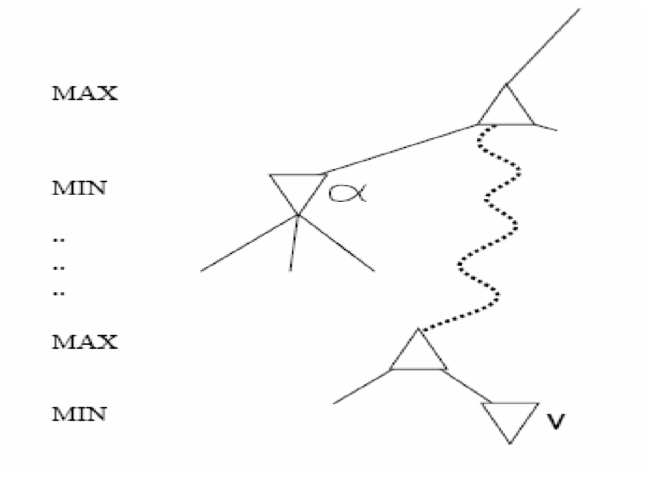




- In other words, the value of the root and hence the minimax decision are independent of the values of the pruned leaves x and y

- Alpha-beta pruning can be applied to trees of any depth, and it is often possible to prune entire subtrees rather than just leaves.

- The general principle is this: consider a node n somewhere in the tree such that Player has a choice of moving to that node.



- If Player has a better choice m either at the parent node of n or at any choice point further up, then n will never be reached in actual play. So once we have found out enough about n to reach this conclusion, we can prune it.

- Alpha-beta pruning gets its name from the following two parameters that describe bounds on the backed-up values that appear anywhere along the path:

  α = the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path for MAX.
  β= the value of the best (i.e., lowest-value) choice we have found so far at any choice point along the path for MIN

- Alpha-beta search updates the values of α and β as it goes along and prunes the remaining branches at a node as soon as the value of the current node is known to be worse than the current α or β value for MAX or MIN respectively.

- The effectiveness of alpha-beta pruning is highly dependent on the order in which the successors are examined. For ex, in figure we could not prune any successors of D at all because the worst successors were generated first. If the third successor had been generated first, we would have been able to prune the other two. This suggests that it might be worthwhile to try to examine first the successors that are likely to be best.

The alpha-beta search algorithm

```
function ALPHA-BETA-SEARCH(state) returns an action
  inputs: state, current state in game

  v ← MAX-VALUE(state, −∞, +∞)
  return the action in SUCCESSORS(state) with value v
```

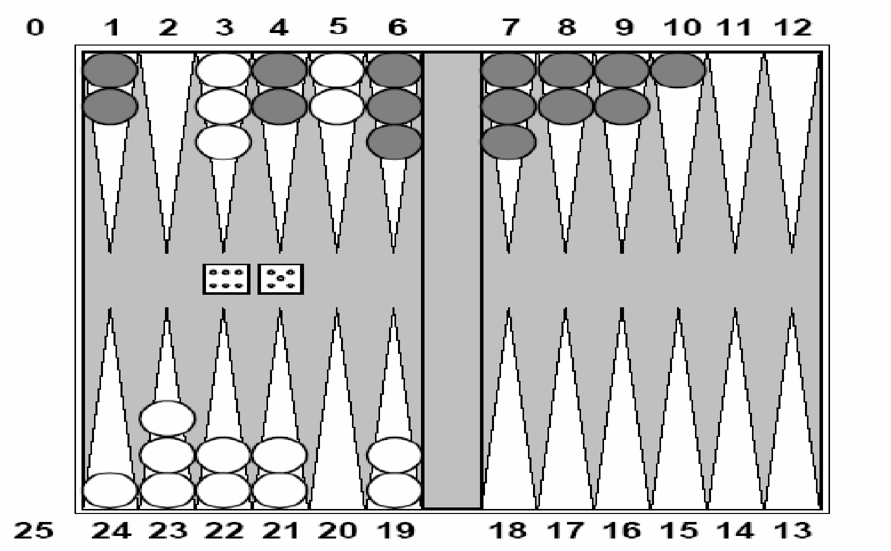
```
function MAX-VALUE(state, α, β) returns a utility value
  inputs: state, current state in game
          α, the value of the best alternative for MAX along the path to state
          β, the value of the best alternative for MIN along the path to state

  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for a, s in SUCCESSORS(state) do
    v ← MAX(v, MIN-VALUE(s, α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v
```

```
function MIN-VALUE(state, α, β) returns a utility value
  inputs: state, current state in game
          α, the value of the best alternative for MAX along the path to state
          β, the value of the best alternative for MIN along the path to state

  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for a, s in SUCCESSORS(state) do
    v ← MIN(v, MAX-VALUE(s, α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

15. STOCHASTIC GAMES

- In real life, there are many unpredictable external events that put us into unforeseen situations.
- Many games mirror this unpredictability by including a random element, such as the throwing of dice.
- In this way, they take us a step nearer reality, and it is worthwhile to see how this affects the decision-making process.
- Backgammon is a typical game that combines luck and skill. Dice are rolled at the beginning of a player’s turn to determine the legal moves. In the backgammon position of figure, for example, white has rolled a 6-5, and has four possible moves.
- Although White knows what his or her own legal moves are, White does not know what Black is going to roll and thus does not know what Black’s legal moves will be.
- That means White cannot construct a standard game tree of the sort we saw in chess and tic-tac-toe.
- A game tree in backgammon must include **chance nodes** in addition to MAX and MIN node Chance nodes are shown as circles in Fig.
- The branches leading from each chance node denote the possible dice rolls, and each is labeled with the roll and the chance that it will occur.

Figure: A typical backgammon position. The goal of the game is to move all one’s pieces off the board.



- There are 36 ways to roll two dice, each equally likely; but because a 6-5 is the same as a 5-6, there are only 21 distinct rolls a 1/18 chance each.
- The next step is to understand how to make correct decisions.
- Obviously, we still want to pick the move that leads to the best position.
- However, the resulting positions do not have definite minimax values.
- Instead, we can only calculate the expected value, where the expectation is taken over all the possible dice rolls that could occur.

This leads us to generalize the minimax value for deterministic games to an expectiminimax value for games with chance nodes.

- Terminal nodes and MAX and MIN nodes work exactly the same way as before; chance nodes are evaluated by taking the weighted average of the values resulting from all possible dice rolls, that is,

  EXPECTIMINIMAX(n) =
  UTILITY(n) if n is a terminal state
  $\text{Max}_{s \in \text{Successors}(n)}$ EXPECTIMINIMAX(s) if n is a MAX node
  $\min_{s \in \text{Successors}(n)}$ EXPECTIMINIMAX(s) if n is a MIN node
  $\sum_{s \in \text{Successors}(n)} P(s).$ EXPECTIMINIMAX(s) if n is a chance node
  Where the successor function for a chance node n simply augments the state of n with each possible dice roll to produce each successor s and P(s) is the probability that that dice roll occurs. These equations can be backed up recursively all the way to the root of the tree, just as in minimax.

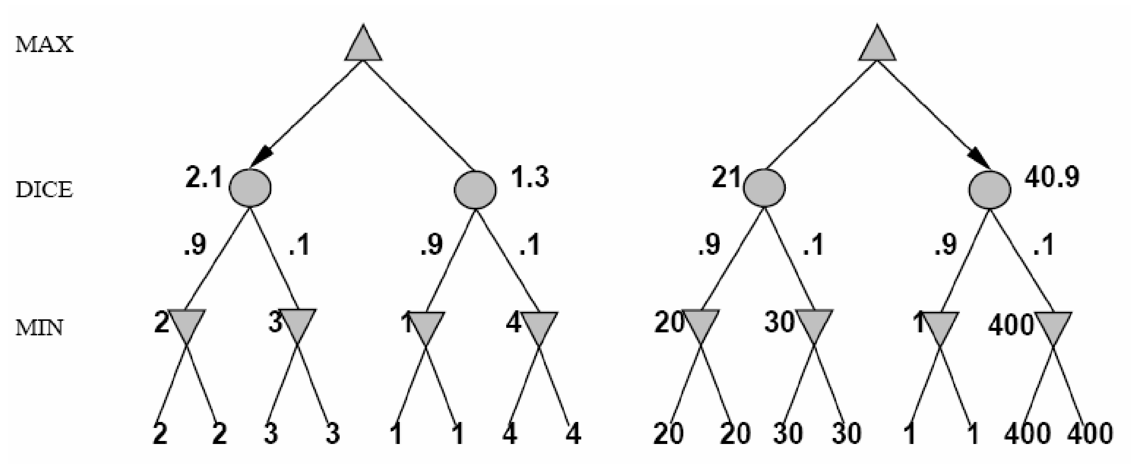
Position evaluation in games with chance nodes

- As with minimax, the obvious approximation to make with expectiminimax is to cut the search off at some point and apply an evaluation function to each leaf.

- One might think that evaluation functions for games such as backgammon should be just like evaluation functions for chess – they just need to give higher scores to better positions.

- But in fact, the presence of chance nodes means that one has to be more careful about what the evaluation values mean.

Figure: An order preserving transformation on leaf values changes the best move.



- Figure shows what happens: with an evaluation function that assigns values [1,2,3,4] to the leaves, move A1 is best; with values [1,20,30,400], move A2 is best. Hence, the program behaves totally differently if we make a change in the scale of some evaluation values.
- It turns out that, to avoid this sensitivity, the evaluation function must be a positive linear transformation of the probability of winning from a position

Complexity of expectiminimax

- If the program knew in advance all the dice rolls that would occur for the rest of the game, solving a game with dice would be just like solving a game without dice, which minimax does in O(b^m) time. Because expectiminimax is also considering all the possible dice-roll sequences, it will take O($b^m n^m$), where n is the number of distinct rolls.

- Even if the search depth is limited to some small depth d , the extra cost compared with that of minimax makes it unrealistic to consider looking ahead very far in most games of chance.
- In backgammon n is 21 and b is usually around 20, but in some situations can be as high as 4000 for dice rolls that are doubles. Three plies is probably all we could manage

- Another way to think about the problem is this: the advantage of alpha-beta is that it ignores future developments that just are not going to happen, given best play. Thus it concentrates on likely occurrences.
- In games with dice, there are no likely sequences of moves, because for those moves to take place, the dice would first have to come out the right way to make them legal.
- This is a general problem whenever uncertainty enters the picture the possibilities are multiplied enormously, and forming detailed plans of action becomes pointless, because the world probably will not play along.
- No doubt it will have occurred to the reader that perhaps something like alpha-beta pruning could be applied to game trees with chance nodes. It turns out that it can.
- The analysis for MIN and MAX nodes is unchanged, but we can also prune chance nodes, using a bit of ingenuity.
- Consider the chance node C in figure and what happens to its value as we examine and evaluate its children.
- Is it possible to find an upper bound on the value of C before we have looked at all its children?

- At first sight, it might seem impossible, because the value of C is the average of its children's values. Until we have looked at all the dice rolls, this average could be anything, because the unexamined children might have any value at all.
- But if we put bounds on the possible values of the utility function, then we can arrive at bounds for the average. For example, if we say that all utility values are between +3 and -3, then the value of leaf nodes is bounded, and in turn we can place an upper bound on the value of the chance node without looking at all its children.

PRATHYUSHA ENGINEERING COLLEGE
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
VI Semester (EVEN 2019-20)
CS8691ARTIFICIAL INTELLIGENCE

UNIT IIIKNOWLEDGE REPRESENTATION 9

First Order Predicate Logic –Prolog Programming –Unification –Forward Chaining-Backward Chaining – Resolution –Knowledge Representation -Ontological Engineering-Categories and Objects –Events -Mental Events and Mental Objects -Reasoning Systems for Categories -Reasoning with Default Information

1. KNOWLEDGE REPRESENTATIONS AND MAPPINGS

In order to solve the complex problems encountered in artificial intelligence, one needs both a large amount of knowledge and some mechanisms for manipulating that knowledge to create solutions to new problems.

A variety of ways of representing knowledge (facts) have been exploited in AI programs. But we are dealing with two different kinds of entities:
• Facts: truths in some relevant world. These are the things we want to represent.
•Representations of facts in some chosen formalism. These are the things we will actually be able to manipulate.

One way to think of structuring these entities is as two levels:
• The knowledge level at which facts are described.
• The symbol level, at which representations of objects at the knowledge level are defined in terms of symbols that can be manipulated by programs.
The model is shown in Fig 2.7. Rather than thinking of one level on top of another, we will focus on facts, on representations, and on the two-way mappings that must exist between them.

We will call these links representation mappings.

- The forward representation mapping maps from facts to representations.
- The backward representation mapping goes the other way from representations to facts.

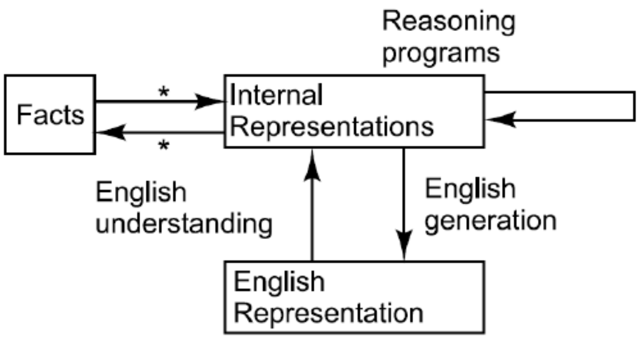


Fig 2.7: Mappings between Facts and Representation

Approaches to knowledge representation

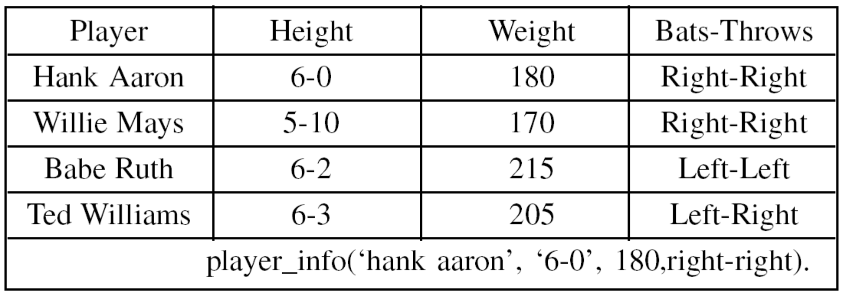
A good system for the representation of knowledge in a particular domain should possess the following four properties:

1. **Representational adequacy** - the ability to represent all the kinds of knowledge that are needed in that domain

2. **Inferential adequacy** – the ability to manipulate the representational structures in such a way as to derive new structures corresponding to new knowledge inferred from old.
3. **Inferential efficiency** – the ability to incorporate into the knowledge structure additional information that can be used to focus the attention of the inference mechanisms in the most promising directions.
4. **Acquisitional efficiency** – the ability to acquire new information easily. The simplest case involves direct insertion, by a person, of new knowledge into the database. Ideally, the program itself would be able to control knowledge acquisition

Simple Relational Knowledge

The simple way to represent declarative facts is as a set of relations of the same sort used in database systems. Figure 2.10 shows an example of such a relational system



Player	Height	Weight	Bats-Throws
Hank Aaron	6-0	180	Right-Right
Willie Mays	5-10	170	Right-Right
Babe Ruth	6-2	215	Left-Left
Ted Williams	6-3	205	Left-Right
player_info('hank aaron', '6-0', 180,right-right).			

Figure 2.10 Simple relational knowldege and a sample fact in Prolog

The relational knowledge of Fig. 2.10 corresponds to a set of attributes and associated values that together describe the objects of the knowledge base

1. The relational knowledge corresponds to a set of attributes and associated values that together describe the objects of the knowledge base.
2. This representation is simple
3. It provides very weak inferential capabilities
4. But knowledge represented in this form may serve as the input to more powerful inference engines.
5. It is not possible even to answer the simple question. "Who is the heaviest player?'
6. But if a procedure for finding the heaviest player is provided, then these facts will enable the procedure to compute an answer.

If, instead, we are provided with a set of rules for deciding which hitter to put up against a given pitcher (based on right- and left-handedness. say), then this same relation can provide at least some of the information required by those rules.

Inheritable Knowledge

The relational knowledge corresponds to a set of attributes and associated values that together describe the objects of the knowledge base. It is possible to augment the basic representation with <u>inference mechanisms that operate on the structure of the representation.</u>

For this to be effective, the structure must be designed so **correspond to the inference mechanism** that are desired. One of the most useful forms of inference is **property inheritance**, in which elements of **specific classes inherit attributes and values from more general classes** in which they are included.

In order to support property inheritance, objects must be organized into classes and classes must be arranged in a generalization hierarchy.
Figure 2.11 shows some additional baseball knowledge inserted into a structure that is so arranged.

- Lines represent attributes.
- Boxed nodes represent objects and values of attributes of objects.

These values can also be viewed as objects with attributes and values, and so on. The arrows on the lines point from an object to its value along the corresponding attribute line. The structure shown in the figure is a slot-and-filler structure. *It may also be called a semantic network or a collection of frames*. In the latter case, each individual frame represents the collection of attributes and values associated with a particular node. Figure 2.12 shows the node for baseball player displayed as a frame.

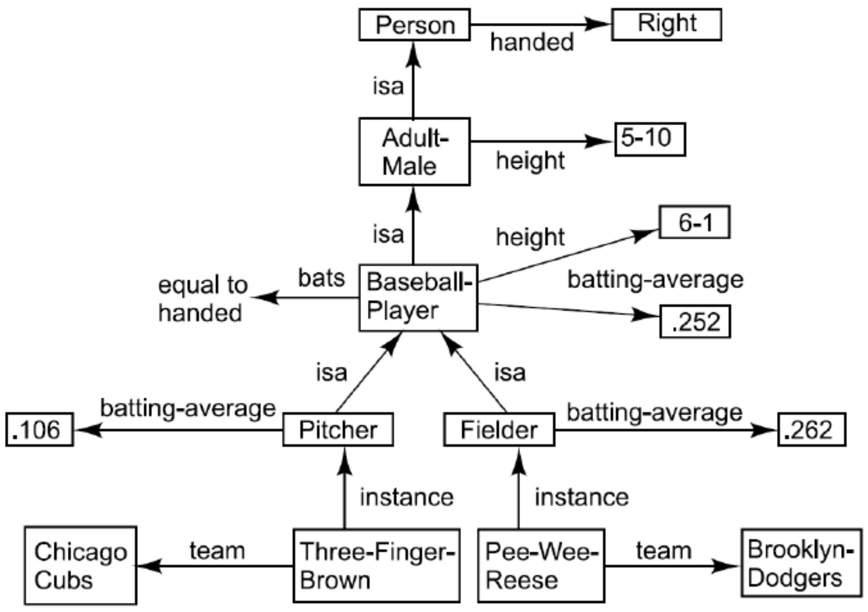


Fig 2.11 Inheritable Knowledge

Baseball-Player

isa:	*Adult-Male*
bats:	(EQUAL *handed)*
height:	6-1
batting-average:	.252

Fig. 2.12 Viewing a Node as a Frame

All of the objects and most of the attributes shown in this example have been chosen to correspond to the baseball domain.

Property inheritance as an inference technique:

The two exceptions to this are the attribute **isa**, which is being used to show class inclusion, and the attribute **instance**, which is being used to show class membership. These two specific attributes provide the basis for property inheritance as an inference technique.

Using this technique, the knowledge base can support retrieval both of facts that have been explicitly stored and of facts that can be derived from those that are explicitly stored. An idealized form of the property inheritance algorithm can be stated as follows:

Algorithm: Property Inheritance

To retrieve a value V for attribute A of an instance object O:

I. Find O in the knowledge base.

2. If there is a value there for the attribute A, report that value.

3. Otherwise, see if there is a value for the attribute instance. If not, then fail.

4. Otherwise, move to the node corresponding to that value and look for a value for the attribute A. If one is found, report it.

5. Otherwise, do until there is no value for the isa attribute or until an answer is found:

   (a) Get the value of the isa attribute and move to that node.

   (b) See if there is a value for the attribute A. If there is, report it.

This procedure is simplistic. It does not say what we should do if there is more than one value of the instance or isa attribute. But it does describe the basic mechanism of inheritance. We can apply this procedure to our example knowledge base to derive answers to the following queries:

- team(Pee-Wee-Reese)= Brooklyn-Dodgers. This attribute had a value stored explicitly in the knowledge base.
- batting-average(Three-Finger Brown) = .106. Since there is no value for batting average stored explicitly for Three Finger Brown, we follow the instance attribute to Pitcher and extract the value stored there. Now we observe one of the critical characteristics of property inheritance, namely that it may produce default values that are not guaranteed to be correct but that represent "best guesses" in the face of a lack of more precise information. In fact, in 1906. Brown's batting average was .204.
- height(Pee-Wee-Reese)= 6-1. This represents another default inference. Notice hem that because we get to it first, the more specific fact about the height of baseball players overrides a more general fact about the height of adult males.
- bats(Three-Finger-Brown) = Right. To get a value for the attribute bats required going up the isa hierarchy to the class Baseball Player. But what we found there was not a value but a rule for computing a value. This rule required another value (that for handed) as input. So the entire process must be begun again recursively to find a value for /tended. This time, it is necessary to go all the way up to Person to discover that the default value for handedness for people is Right Now the rule for bats can he applied. producing the result Right. In this case, that turns out to be wrong. since Brown is a switch hitter (i.e., he can hit both left-and right-handed).

Inferential Knowledge

Property inheritance is a powerful form of inference, but it is not the only useful form. Sometimes all the power of traditional logic (and sometimes even more than that) is necessary to describe the inferences that are needed. Figure 2.13 shows two examples of the use of first-order predicate logic to represent additional knowledge about baseball.

$$\forall x : Ball(x) \wedge Fly(x) \wedge Fair(x) \wedge Infield\text{-}Catchable\,(x) \wedge$$
$$Occupied\text{-}Base(First) \wedge Occupied\text{-}Base(Second) \wedge (Outs < 2) \wedge$$
$$\neg[Line\text{-}Drive(x) \vee Attempted\text{-}Bt,(x)]$$
$$\rightarrow Infield\text{-}Fly(x)$$
$$\forall x,y : Batter(x) \wedge batted(x,\ y) \wedge Infield\text{-}Fly(y) \rightarrow \mathrm{Out}(x)$$

Fg. 2.13 Inferential Knowledge

The required inference procedure now is one that implements the standard logical rules of inference. There are many such procedures, some of which reason forward from given facts to conclusions, others of which reason backward from desired conclusions to given facts.
One of the most commonly used of these procedures is resolution, which exploits a proof by contradiction strategy.

Procedural Knowledge
Procedural knowledge can be represented in programs in many ways.
The most common way is simply as code (in some programming language such as LISP) for doing something. The machine uses the knowledge when it executes the code to perform a task. This way of representing procedural knowledge is not efficient with respect to the properties of *inferential adequacy* (because it is very difficult to write a program that can reason about another program's behavior) and *acquisitional efficiency* (because the process of updating and debugging large pieces of code becomes unwieldy).

1. As an extreme example, compare the representation of the way to compute the value of bars shown in Fig. 2.12 to one in LISP shown in Fig. 2.14. *Although the LISP one will work given a particular way of storing attributes and values in a list, it does not lend itself to being reasoned about in the same straightforward way as the representation of Fig. 4.6 does*. The LISP representation is slightly more powerful since it makes explicit use of the name of the node whose value for handed is to be found. But if this matters, the simpler representation can be augmented to do this as well.

```
 Baseball-Player
isa:            Adult-Male
bats:           (lambda 00
                        (prog 0
                               L I
                                      (cord ((caddr x) (return (caddr x)))
                                              (t (setq x (eval (eadr x)))
                                                     (cond (x (go L I))
                                                             (return nil)))))))
height:                 6-1
basing-average::       .252
```

Fig. 2.14 Using LISP Code to Define a Value

Because of this difficulty in reasoning with LISP, attempts have been made to find other ways of representing procedural knowledge so that it can relatively easily be manipulated both by other programs and by people.

The most commonly used technique for representing procedural knowledge in AI programs is the use of production rules. Figure 2.15 shows an example of a production rule that represents a piece of operational knowledge typically possessed by a baseball player.

Production rules, particularly ones that are augmented with information on how they are to be used, are more procedural than are the other representation methods.

But making a clean distinction between declarative and procedural knowledge is difficult. Although at an intuitive level such a distinction makes some sense, at a formal level it disappears. The important difference is in how the knowledge is used by the procedures that manipulate it.

```
If:     ninth inning, and
        score is close, and
        less than 2 outs, and
        first base is vacant, and
        batter is better hitter than next batter,
Then:  walk the batter.
```

Fig. 2.15 Procedural Knowledge as Rules

2. ISSUES IN KNOWLEDGE REPRESENTATION

Several issues that cut across all of them:

- Are any attributes of objects so basic that they occur in almost every problem domain? If there are, we need to make sure that they are handled appropriately in each of the mechanisms we propose. If such attributes exist, what are they?
- Are there any important relationships that exist among attributes of objects?
- At what level should knowledge be represented? Is there a good set of primitives into which all knowledge can be broken down? Is it helpful to use such primitives?
- How should sets of objects be represented?
- Given a large amount of knowledge stored in a database, how can relevant parts be accessed when they are needed?

2.1 Important Attributes

*Are **any attributes of objects so basic** that they occur in almost every problem domain*? If there are, we need to make sure that they are handled appropriately in each of the mechanisms we propose. If such attributes exist, what are they?

There are two attributes that are of very general significance: instance and isa.

- These attributes are important because they support property inheritance.
- They represent class membership and class inclusion and that *class inclusion is transitive*.
- In slot-and-filler systems, such as those attributes are usually represented explicitly
- These relationships may be represented this way or they may be represented implicitly by a set of predicates describing particular classes.

2.2 Relationships among Attributes

• *Are there any **important relationships** that exist among attributes of objects?*
There are four such properties:

- Inverses
- Existence in an isa hierarchy
- Techniques for reasoning about values
- Single-valued attributes

Inverses

Entities in the world are related to each other in many different ways.
But as soon as we decide to describe those relationships as attributes, we focus on one object and look for **binary relationships** between it and others.
Each of these was shown with a **directed arrow**, originating at the object that was being described and terminating at the object representing the value of the specified attribute.

But if object representing the value is considered, then there is still a relationship between the two entities, although it is a different one since the original relationship was not symmetric (although some relationships, such as sibling. are).

In many cases, it is important to represent this other view of relationships.
There are two ways to represent the relationship.

1. The first is to represent both relationships in a single representation that ignores focus. Logical representations are usually interpreted as doing this.
   For example, the assertion:

team(Pee-Wee-Reese, Brooklyn-Dodgers)

can equally easily be interpreted as a statement about Pee Wee Reese or about the Brooklyn Dodgers. How it is actually used depends on the other assertions that a system contains.

2. The second approach is to use attributes that focus on a single entity but to use them in pairs, **one the inverse of the other**. In this approach, we would represent the team information with two attributes:
   - one associated with Pee Wee Reese:
     team = Brooklyn-Dodgers
   - one associated with Brooklyn Dodgers:
     team-members = Pee-Wee-Reese,...

This is the approach that is taken in semantic net and frame-based systems. When it is used, it is usually accompanied by a knowledge acquisition tool that guarantees the consistency of inverse slots by forcing them to be declared and then checking each time a value is added to one attribute that the corresponding value is added to the inverse.

An Isa Hierarchy of Attributes

Just as there are classes of objects and specialized subsets of the classes, there are attributes and specializations of attributes.

They support inheriting information about such things as constraints on the values that the attribute can have and mechanisms for computing those values.

Techniques for Reasoning about Values

Sometimes values of attributes are specified explicitly when a knowledge base is created. But often the reasoning system must reason about values it has not been given explicitly.

Several kinds of information can play a role in this reasoning, including:

• Information about the **type of the value**. For example, the value of height must be a number measured in a unit of length.

• **Constraints on the value**, often stated in terms of related entities. For example, the age of a person cannot be greater than the age of either of that person's parents.

• **Rules for computing the value** when it is needed. We showed an example of such a rule in Fig. 4.5 for the bats attribute. These rules are called backward rules. Such rules have also been called *if-needed rules*.

• **Rules that describe actions that should be taken if a value ever becomes known**. These rules are called lanyard rules, or sometimes *if-added rules*.

Single-Valued Attributes

This attribute is one that is **guaranteed to take a unique value**.

For example, a baseball player can, at any one time, have only a single height and be a member of only one team.

If there is already a value present for one of these attributes and a different value is asserted, then one of two things has happened. **Either a change has occurred** in the world or there is now a **contradiction in the knowledge base** that needs to be resolved. Knowledge-representation systems have taken several different approaches to providing support for single-valued attributes, including:

• **Introduce an explicit notation for temporal interval**. If two different values are ever asserted for the same temporal interval, signal a contradiction automatically.

• *Assume that the only temporal interval that is of interest is now*. So if a new value is asserted, replace the old value.

• *Provide no explicit support*. Logic-based systems are in this category. But in these systems, knowledge-base builders can add axioms that state that if an attribute has one value then it is known not to have all other values.

2.4.3 Choosing the Granularity of Representation

At what level should knowledge be represented? Is there a good set of primitives into which all knowledge can be broken down? Is it helpful to use such primitives?

A brief example illustrates the problem. Suppose we are interested in the following fact:

John spotted Sue.

We could represent this as

spotted(agent(John)
object(Sue))

Such a representation would make it easy to answer questions such as: Who spotted Sue! But now suppose we want to know:

Did John see Sue?

The obvious answer is "yes." but given only the one fact we have, we cannot discover that answer. We multi, of course, add other facts, such as

spotted(x,y) →saw (x, y)

We could then infer the answer to the question. An alternative solution to this problem is to represent the fact that spotting is really a special type of seeing explicitly in the representation of the fact. We might write something such as

Saw(agent(John), object(Sue), timespan(briefly))

In this representation, we have broken the idea of spotting apart into more primitive concepts of seeing and timespan.

Using this representation, the fact that John saw Sue is immediately accessible. But the fact that he spotted her is more difficult to get to. The major advantage of converting all statements into a representation in terms of a *small set of primitives is that the rules that are used to derive inferences from that knowledge need be written only in terms of the primitives* rather than in terms of the many ways in which the knowledge may originally have appeared. Thus what is really being argued for is simply some sort Of canonical form. Several AI programs are based on knowledge bases described in terms of a small number of low-level primitives.

There are several arguments against the use of low-level primitives. One is that simple high-level facts may require a **lot of storage when broken down into primitives**.

Much of that storage is really wasted since the low-level rendition of a particular high- level concept will appear many times, once for each time the high-level concept is referenced.

If knowledge is initially presented to the system in a relatively high-level form, such as English, then substantial work must be done to reduce the knowledge into primitive form. This detailed primitive representation may be unnecessary.

For the sake of efficiency, *it may be desirable to store these things at a very high level and then to analyze in detail only those inputs that appear to be important.*

A third problem with the use of low-level primitives is that in many domains, it is not at all clear what the primitives should be. And even in domains in which there may be an obvious set of primitives, there may not be enough information present in each use of the high-level constructs to enable them to be convened into their primitive components.

For example, given just the fact

John broke the window.

a program would not be able to decide if John's actions consisted of the primitive sequence:

1. Pick up a hard object.
2. Hurl the object through the window

or the sequence:

1. Pick up a hard object.
2. Hold onto the object while causing it to crash into the window.

or the single action:

1. Cause hand (or foot) to move fast and crash into the window.

or the single action:

1. Shut the window so hard that the glass breaks.

As these examples have shown, the problem of choosing the correct granularity of representation for a particular body of knowledge is not easy. Clearly, the lower the level we choose, the less inference required to reason with it in some cases. The answer for any particular task domain must come to a large extent from the domain itself—to what use is the knowledge to be put?

2.3 Representing Sets of Objects

How should sets of objects be represented?

It is important to be able to represent sets of objects for several reasons.

There are some properties that are true of sets that are not true of the individual members of a set. As examples, consider the assertions that are being made in the sentences "There are more sheep than people in Australia" and "English speakers can be found all over the world."

There are three obvious ways in which sets may be represented.

1. The simplest is just by a **name**. This is essentially when we used the node named Baseball-Player in our semantic net and when we used predicates such as Ball and Batter in our logical representation. This simple representation does make it possible to associate predicates with sets.
2. There **are two ways to state a definition of a set and its elements**.
   a. The first is to list the members. Such a specification is called an **extensional** definition.
      For example, an extensional description of the set of our sun's planets on which people live is {/Earth}
   b. The second is to provide a rule that, when a particular object is evaluated, returns true or false depending on whether the object is in the set or not. Such a rule is called an intensional definition. An **intensional** description is {x :sun-planent(x)) ∧ human-inhabited(x)}

Thus, while it is trivial to determine whether two sets an identical if extensional descriptions are used, it may be very difficult to do so using intensional descriptions.

Intensional representations have two important properties that extensional ones lack.

1. The first is that they can be used to describe **infinite sets** and sets not all of whose elements are explicitly known. Thus we can describe intensionally such sets as prime numbers (of which there an infinitely many) or kings of England (even though we do not know who all of them are or even how many of them there have been).
2. The second thing we can do with intensional descriptions is to allow them to **depend on parameters that can change**, such as time or spatial location. If we do that, then the actual set that is represented by the description will change as a function of the value of those parameters.

2.4 Finding the Right Structures as Needed

Given a large amount of knowledge stored in a database, how can ***relevant parts be accessed*** *when they are needed?*

For example, suppose we have a script that describes the typical sequence of events in a restaurant. This script would enable us to take a text such as

> John went to Steak and Alt last night. He ordered a large steak. paid his bill, and left
> and answer "yes" to the question
>
> Did John eat dinner last night?

Notice that nowhere in the story was John's eating anything mentioned explicitly. But the fact that when one goes to a restaurant one eats will be contained in the restaurant script. If we know in advance to use the restaurant script, then we can answer the question easily. But in order to be able to reason about a variety of things, a system must have many scripts for everything from going to work to sailing around the world. How will it select the appropriate one each time? For example, nowhere in our story was the word -restaurant" mentioned. In fact, in order to have access to the right structure for describing a particular situation, it is necessary to solve all of the following problems

- How to perform an initial selection of the most appropriate structure.
- How to fill in appropriate details from the current situation.
- How to find a better structure if the one chosen initially turns out not to be appropriate.
- What to do if none of the available structures is appropriate.
- When to create and remember a new structure.

There is no good. General purpose method for solving all these problems. Some knowledge-representation techniques solve some of them.

1. **Selecting an Initial Structure**

Selecting candidate knowledge structures to match a particular problem-solving situation is a hard problem; Consider each major concept as a pointer to all of the structures (such as scripts) in which it might be involved. This may produce several sets of prospective structures.

2. Locate one major clue in the problem description and use it to select an initial structure. As other clues appear, use them to refine the initial selection or to make a completely new one if necessary.
3. Select the fragments of the current structure that do correspond to the situation and match them against candidate alternatives. Choose the best match. If the current structure was at all close to being appropriate, much of the work that has been done to build substructures to fit into it will be preserved.

   • Refer to specific stored links between structures to suggest new directions in which to explore.

   • If the knowledge structures are stored in an isa hierarchy, traverse upward in it until a structure is found that is sufficiently general that it does not conflict with the evidence.
4. Either use this structure if it is specific enough to provide the required knowledge or consider creating a new structure just below the matching one.

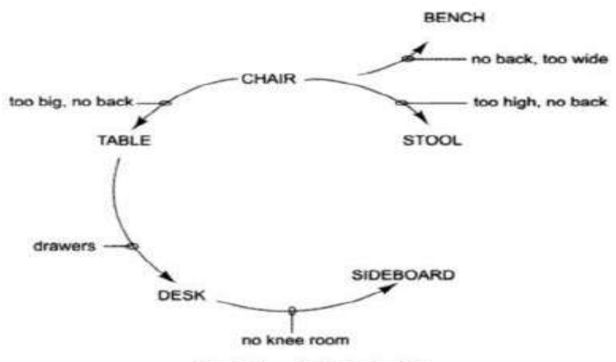


Fig. 4.11 *A Similarity Net*

2.5 The Frame Problem

Another issue concerns how to represent efficiently sequences of problem states that arise from a search process. For complex ill-structured problems, this can be a serious matter.

Consider the world of a household robot. There are many objects and relationships in the world, and a state description must somehow include facts like on(Plant12,Table34), under(Table34, Window13), and in(Table34, Room 15).

One strategy is to store each state description as a list of such facts. But what happens during the problem-solving process if each of those descriptions is very long? Most of the facts will not change from one state to another, yet each fact will be represented once at every node, and we will quickly run out of memory.

Furthermore, we will spend the majority of our time creating these nodes and copying these facts—most of which do not change often—from one node to another.

For example, in the robot world, we could spend a lot of time recording above(Ceiling. Floor) at every node. All of this is, of course, in addition to the real problem of figuring out which facts should be different at each node.

This whole problem of representing the facts that change as well as those that do not is known as the frame problem. In some domains, the only hard part is representing all the facts. In others, though, figuring out which ones change is nontrivial. For example, in the robot world, there might be a table with a plant on it under the window. Suppose we move the table to the center of the room. We must also infer that the plant is now in the center of the mom too but that the window is not.

To support this kind of reasoning, some systems make use of an explicit set of axioms called frame axioms, which describe all the things that do not change when a particular operator is applied in state n to produce state n + 1. (The things that do change must be mentioned as part of the operator itself.) Thus, in the robot domain, we might write axioms such as

color(x,y,s1) ∧ move(x, s1,S2)→ color(x,y, S2)

which can be read as. "If x has color y in state si and the operation of moving x is applied in state si to produce state S2, then the color of x in s2 is still y. This solves the problem of the wasted space and time involved in copying the information for each node. And it works fine until the first time the search has to backtrack.

There are two ways this problem can be solved:

- • Do not modify the initial state description at all. At each node, store an indication of the specific changes that should be made at this node.
- • Modify the initial state description as appropriate, but also record at each node an indication of what to do to undo the move should it ever be necessary to backtrack through the node. Then, whenever it is necessary to backtrack, check each node along the way and perform the indicated operations on the Mate description.

3 **PROPOSITIONAL LOGIC**

Propositional logic is appealing *because it is simple to deal with and a decision procedure for it exists*.

We can easily represent real-world facts as logical propositions written as well formulas (wff's) in propositional logic.

Using these propositions, we could, conclude from the fact that it is mining the fact that it is not sunny. But very quickly we run up against the limitations of propositional logic.

Suppose we want to represent the obvious fact stated by the classical sentence

It is raining.
RAINING
It is sunny.
SUNNY
It is windy.
WINDY
It is raining, then it is not sunny.
RAINING SUNNY

Fig. 2.16 Some Simple Facts in Prepositional Logic

Socrates is a man.

We could write:

SOCRATESMAN

But if we also wanted to represent

Plato is a man.

we would have to write something such as:

PLATOAMAN

which would be a totally separate assertion, and we would not be able to draw any conclusions about similarities between Socrates and Plato. It would be much better to represent these facts as:

MAN(SOCRATES) MAN(PLATO)

Now the structure of the representation reflects the structure of the knowledge itself. But to do that, we need to be able to use *predicates applied to arguments*. We are in even more difficulty if we try to represent the equally classic sentence

All men are mortal. We could represent this as: MORTALMAN

But that fails to capture the relationship between any individual being a man and that individual being a mortal.

To do that, we really need <u>variables and quantification</u> unless we are willing to write separate statements about the mortality of every known man.

But a major motivation for choosing to use logic

1. A good way of reasoning with that knowledge is available.
2. Determining the validity of a proposition in propositional logic is straightforward.
3. It provides a way of deducing new statements from old ones. But, it does not possess a decision procedure. There do exist procedures that will find a proof of a proposed theorem if indeed it is a theorem. But these procedures are not guaranteed to halt if the proposed statement is not a theorem. F*irst-order predicate logic is not decidable, it is semidecidable.*

A simple such procedure is to use the *rules of inference to generate theorem's* from the axioms in some orderly fashion, testing each to see if it is the one for which a proof is sought.

- Propositional logic is a declarative language because its semantics is based on a truth relation between sentences and possible worlds.
- It is also has sufficient expressive power to deal with partial information, using disjunction and negation.
- Propositional logic has a third property that is desirable in representation languages, namely **compositionality**. In a compositional language, the meaning of a sentence is a function of the meaning of its parts.

For example, "$S_{1,4}$ ^$S_{1,2}$" is related to the meaning of "$S_{1,4}$" and "$S_{1,2}$".

<u>BNF Grammar Propositional Logic</u>

$$
\begin{array}{lll}
\mathit{Sentence} & \rightarrow & \mathit{AtomicSentence} \mid \mathit{ComplexSentence} \\
\mathit{AtomicSentence} & \rightarrow & \texttt{True} \mid \texttt{False} \mid \texttt{P} \mid \texttt{Q} \mid \texttt{R} \mid \ldots \\
\mathit{ComplexSentence} & \rightarrow & (\mathit{Sentence}\,) \\
 & & \mid \mathit{Sentence}\ \mathit{Connective}\ \mathit{Sentence} \\
 & & \mid \neg\, \mathit{Sentence} \\
\mathit{Connective} & \rightarrow & \wedge \mid \vee \mid \Rightarrow \mid \Leftrightarrow
\end{array}
$$

Drawback in propositional logic

Propositional logic lacks the expressive power to describe an **environment with many objects** concisely. For example, we were forced to write a separate rule about breezes and pits for each square, such as

$B_{1,1} <=> (P_{1,2} \lor P_{2,1})$

Programming languages

- Programming languages such as C++ or Java or Lisp are by far the largest class of formal languages in common use.
- Data structures within programs can represent facts; for example, a program could use a 4 X 4 arrays to represent the contents of the wumpus world.

Drawback in data structures:

1. Programming languages lack is any general mechanism for deriving facts from other facts
2. Programs can store a single value for each variable. It is difficult to represent, for example "There is a pit in square[2,2] or [3,1]" or "If the Wumpus is in [1,1] then he is not in [2, 2]".

Natural languages

The natural languages are very expressive

In natural languages, it is easy to say "squares Adjacent to pits are breezy", even for a grid of Infinite size.

Natural languages like English or German:

• are far more expressive than propositional logic

• serve as medium for communication

• depend heavily on the context

• can be ambiguous

When a speaker points and says "Look!" the listener comes to know that, say, Superman has finally appeared over the rooftops. Yet we would not want to say that the sentence "Look!" encoded that fact.

Rather, **the meaning of the sentence depends both on the sentence itself and on the context in which the sentence was spoken**. Clearly, one could not store a sentence such as "Look!" in a knowledge base and expect to recover its meaning without also storing a representation of the context – which raises the question of how the context itself can be represented.

The syntax of natural language, the most obvious elements are

1. *Nouns and noun phrases* that refer to **objects,** and
2. *Verbs and verb phrase* that refer to **relations** among objects.
3. Some of these relations are **functions**— relations in which there is only one "value" for a given "input."

It is easy to start listing examples of objects, properties, relations, and functions:

- **Objects**: people, houses, numbers, theories, Ronald McDonald, colors, baseball games, wars, centuries
- **Relations**: brother of, bigger than, inside, part of, has color, occurred after, owns...
- **Properties**: red, round, bogus, prime, multistoried...
- **Functions**: father of, best friend, third inning of, one more than ...

Some examples follow:

• "One plus two equals three"

Objects: one, two, three, one plus two;

Relation: equals;

Function: plus. (One plus two is a name for the object that is obtained by applying the function plus to the objects one and two. Three is another name for this object.)

• "Squares neighboring the wumpus are smelly."

Objects: wumpus, square;

Property: smelly;

Relation: neighboring.

- "Evil King John ruled England in 1200."

Objects: John, England, 1200;

Relation: ruled;

Properties: evil, king.

Inference Rules for Propositional logic:

- The propositional logic has seven inference rules.
- Inference means conclusion reached by reasoning from data or premises; speculation.
- A procedure which combines known facts to produce ("infer") new facts.
- **Logical inference** is used to create new sentences that logically follow from a given set of predicate calculus sentences (KB).
- An inference rule is **sound** if every sentence X produced by an inference rule operating on a KB logically follows from the KB. (That is, the inference rule does not create any contradictions)
- An inference rule is **complete** if it is able to produce every expression that logically follows from (is entailed by) the KB. (Note the analogy to complete search algorithms.)
- Here are some examples of sound rules of inference

A rule is sound if its conclusion is true whenever the premise is true

Rule	Premise	Conclusion
Modus Ponens	$A, A \rightarrow B$	B
And Introduction	A, B	$A \wedge B$
And Elimination	$A \wedge B$	$A \vee B$
Or Introduction	A, B	A
Double Negation	$\neg\neg A$	A
Unit Resolution	$A \vee B, \neg B$	A

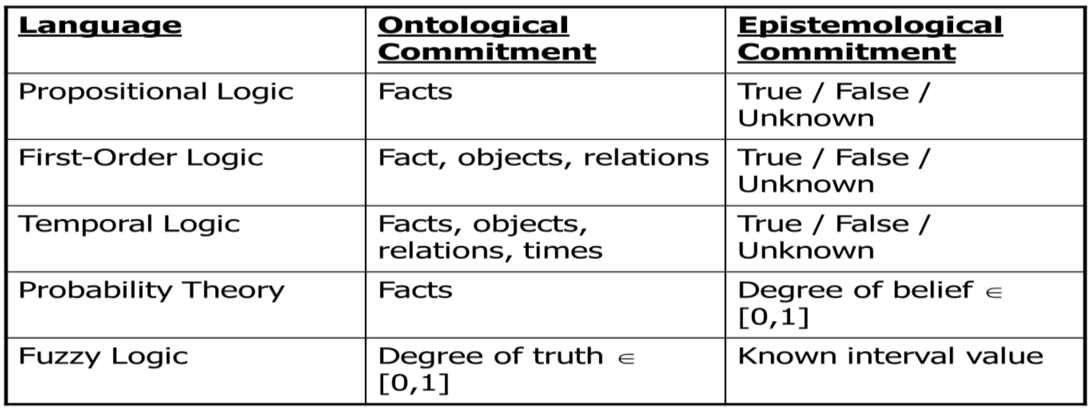
Resolution	$A \vee B, \neg B \vee C$	$A \vee C$

4. PREDICATE LOGIC

First-order logic (PREDICATE LOGIC), is sufficiently expressive to represent a good deal of our commonsense knowledge. It also either subsumes or forms the foundation of many other representation languages.

- First-order logic (FOL) is built around objects and relations
- Has greater expressive power than propositional logic
- It can also express facts about some or all objects in the universe.

- First-Order Logic assumes that the world contains:
  - **Objects** (E.g. people, houses, numbers, theories, colors, football games, wars, centuries, …)
  - **Relations** (E.g. red, round, prime, bogus, multistoried, brother of, bigger than, inside, part of, has color, occurred after, owns, comes between, …)
  - **Functions** (E.g. father of, best friend, third quarter of, one more than, beginning of,..)
- **Epistemological commitments** define the possible states of knowledge that a logic allows with respect to each fact



Language	Ontological Commitment	Epistemological Commitment
Propositional Logic	Facts	True / False / Unknown
First-Order Logic	Fact, objects, relations	True / False / Unknown
Temporal Logic	Facts, objects, relations, times	True / False / Unknown
Probability Theory	Facts	Degree of belief $\in$ [0,1]
Fuzzy Logic	Degree of truth $\in$ [0,1]	Known interval value

The primary difference between first-order logic and propositional logic lies in the **Ontological commitments** made by each language- that is, what it assumes about the nature of *reality*.

Ontological commitments

- **Propositional logic** assumes that there are facts that either hold or do not in the world. Each fact can be in one of two states: true or false.
- **First-order logic** assumes more: namely, that the world consists of objects with certain relations between them that do or do not hold.
- **Temporal logic** assumes that the world is ordered by a set of time points or intervals, and includes built-in mechanisms for reasoning about time.

Epistemological commitments have to do with the possible states of *knowledge* an agent can have using various types of logic.

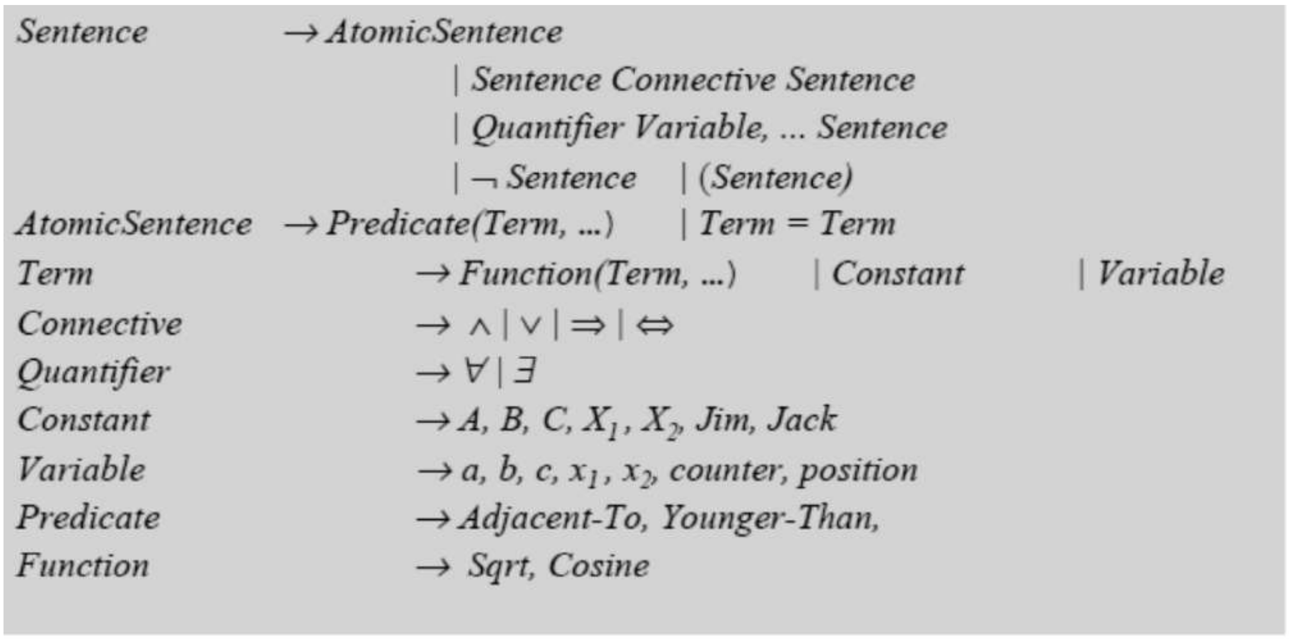
- **Propositional logic & First-order logic** have three possible states of belief regarding any sentence.
- **Probability theory**, can have any *degree* of belief, ranging from 0 (total disbelief) to 1 (total belief).
  - o For example, a probabilistic wumpus-world agent might believe that the wumpus is in [1,3] with probability 0.75.

- **Fuzzy logic** can have degrees of belief in a sentence, and also allow *degrees of truth:* a fact need not be true or false in the world, but can be true to a certain degree.

For example, "Vienna is a large city" might be true only to degree 0.6. The ontological and epistemological commitments of various logics are summarized

Syntax and Semantics of First-Order Logic

Bakus Naur Form (BNF) for FOL



Sentence $\rightarrow$ *AtomicSentence*
| *Sentence Connective Sentence*
| *Quantifier Variable, ... Sentence*
| $\neg$ *Sentence* | *(Sentence)*

AtomicSentence $\rightarrow$ *Predicate(Term, ...)* | *Term = Term*

Term $\rightarrow$ *Function(Term, ...)* | *Constant* | *Variable*

Connective $\rightarrow \wedge \mid \vee \mid \Rightarrow \mid \Leftrightarrow$

Quantifier $\rightarrow \forall \mid \exists$

Constant $\rightarrow$ *A, B, C,* X_1, X_2, *Jim, Jack*

Variable $\rightarrow$ *a, b, c,* x_1, x_2, *counter, position*

Predicate $\rightarrow$ *Adjacent-To, Younger-Than,*

Function $\rightarrow$ *Sqrt, Cosine*

First-order logic is also called (first-order) predicate logic and predicate calculus

Models for first-order logic

- The models of a logical language are the **formal structures that constitute the possible worlds** under consideration.
- Models for propositional logic are just sets of truth values for the proposition symbols.
- The domain of a model in FOL is the *set of objects or domain elements* that it contains
- It must be *nonempty,* every possible world must contain at least one object
- It doesn't matter what these objects are
- In the following examples, we use named objects, i.e., we refer to objects using their names

Relations:

The objects in the model may be related in various ways. In the figure, Richard and John are brothers. Formally speaking a relation is just the set of tuples of objects that are related.

1. The "brother" relation in this model is the set {<Richard the Lionheart, King John>, <King John, Richard the Lionheart>}
2. The "on head" relation in this model is the set {<the crown, King John>}
3. The "person" property (unary relation) is the set {<Richard the Lionheart>, <King John>}

Properties:

1. The model also contains **unary relations, or properties: the "person" property** is true of both Richard and John;

2. he **“king” property** is the only of John; and
3. **the “crown” property** is true only of the crown.

Certain kinds of relationships are best considered as functions, in that a given object must be related to exactly one object in this way. For example, each person has one left leg, so the model has a **unary “left leg” function** that includes the following mappings:
(Richard the Lionheart) → Richard’s left leg
(King John) → John’s left leg

Components of FOL

i. **Symbols**

The basic syntactic elements of first-order logic are the symbols that stand for objects, relations, and functions.

The symbols, therefore, come in three kinds:
Constant symbols which stands for objects; ex, Marcus, Caesar
Predicate symbols which stands for relations ex, Person, Man, loyalto
Function symbols which stands for functions ex Left_leg

ii. **Interpretations**

The semantic must relate sentences to models in order to determine truth. For this to happen, we need an interpretation that specifies exactly which objects, relations and functions are referred to by the constant, predicate and function symbols.

One possible interpretation for our example – which we will call the intended interpretation – is as follows:

- Richard refers to Richard the Lionheart and John refers to the evil King John
- Brother refers to the brotherhood relation, that is , the set of tuples of objects
- LeftLeg refers to the “left leg” function

Not all the objects need have name – for example, the intended interpretation does no name the crown or the legs.

It is also possible for an object to have several names; there is an interpretation under which both Richard and John refer to the crown.

In propositional logic, it is perfectly possible to have a model in which cloudy and sunny are both true; it is the job of the knowledge base to rule out models that are inconsistent with our knowledge.

iii. **Terms**

A term is a logical expression that refers to an object. Constant symbols are therefore terms. Sometimes, it is more convenient to use an expression to refer to an object.
For example, in English we might use the expression "King John's left leg" rather than giving a name to his leg. This is what function symbols are for: instead of using a constant symbol, we use *LeftLegOf (John).*

Complex term is formed by a function symbol followed by a parenthesized list of terms as arguments to the function symbol.

Thus *LeftLegOf* function symbol might refer to the following functional relation:
{ (King John, King John's left leg), (Richard the Lionheart, Richard's left leg)}

and if *King John* refers to King John, then *LeftLegOf (King John)* refers to King John's left leg.

iv. Atomic sentences

An atomic sentence is formed from a predicate symbol followed by a parenthesized list of terms.

For example, *Brother(Richard, John)* states, under the interpretation given before, that Richard the Lionheart is the brother of King John. Atomic sentences can have arguments that are complex terms: *Married(FatherOf (Richard),MotherOf (John))* states that Richard the Lionheart's father is married to King John's mother (again, under a suitable interpretation).

An atomic sentence is true if the relation referred to by the predicate symbol holds between the objects referred to by the arguments relation holds just in case the tuple of objects is in the relation! The truth of a sentence therefore depends on both the interpretation and the world.

v. Complex sentences

Logical connectives are used to construct more complex sentences, just as in propositional calculus. The semantics of sentences formed using logical connectives is identical to that in the propositional case.

For example:
• *Brother(Richard, John)* Λ *Brother(John, Richard)* is true just when John is the brother of Richard and Richard is the brother of John.
• *Older(John,* 30) V *Younger(John,* 30) is true just when John is older than 30 or John is younger than 30.
• *Older(John,* 30) => ¬*Younger(John,* 30) states that if John is older than 30, then he is not younger than 30
• ¬*Brother(Robin, John)* is true just when Robin is not the brother of John.

vi. Quantifiers

Quantifiers let us express properties of entire collections of objects, instead of enumerating the objects by name. First-order logic contains two standard quantifiers, called *universal* and *existential.*
Universal quantification: $\forall x$
Existential quantification: $\exists x$

a. **Universal quantification** (V)

"All kings are persons" is written in first-order logic as

$\forall$ x King(x) => Person(x)
Thus, the sentence says, "For all x, if x is a king, then x is a person". The symbol x is called a **variable.**

The variable is a term all by itself, and as such can also serve as the argument of a function – for example, LeftLeg(x).
A term with no variables is called a ***ground term.***

Intuitively, the sentence Vx P, where P is any logical expression, says that P is true for every object x. More precisely, Vx P is true in a given model under a given interpretation if P is true in all possible **extended interpretations** constructed from the given interpretation, where each extended interpretation specifies a domain element to which x refers.

The universally quantified sentence $\forall$ x King(x) => Person(x) is true under the original interpretation if the sentence King(x) => Person(x) is true in each of the five extended interpretations.

b. **Existential quantification** ($\exists$)

Universal quantification makes statements about every object. Similarly, we can make a statement about *some object in the universe without naming it, by using an existential quantifier.*

To say, for example, that King John has a crown on his head, we write

$\forall$x$\exists$ y person(x)$\wedge$person(y)$\wedge$father(y,x)

$\exists$ is pronounced "There exists ..." In general, $\exists$ *x P* is true if *P* is true for *at least one* object x.

More precisely, $\exists$ *x P is true in a* given model under a given interpretation if P is true in at least one extended interpretation that assigns x to a domain element.

Nested quantifiers

We will often want to express more complex sentences using multiple quantifiers. The simplest case is where the quantifiers are of the same type.

For example, "Brothers are siblings" can be written as
$\forall$ *x* $\forall$ *y Brother(x,y)* => Sibling(x,y)
Consecutive quantifiers of the same type can be written as on quantifier with several variables. For example, to say that siblinghood is a symmetric relationship, we can write
$\forall$ *x,y Sibling(x,y) => Sibling(y,x)*
In other cases we will have mixtures. "Everybody loves somebody" means that for every person, there is someone that person loves:

$\forall$ *x* $\exists$ y *Loves(x,y)*

On the other hand, to say "There is someone who is loved by everyone" we write

$\exists$ y $\forall$ *x Loves(x,y)*

The order of quantification is therefore very important. It becomes clearer if we put in parentheses. $\forall$ x ($\exists$ y Loves*(x,y)) says that everyone has a particular property, namely, the property that somebody loves them.*
On the other hand $\exists$ x ($\forall$ y Loves*(x,y)* says that there is *some* objectin the world that has a particular property, namely the property of being loved by everybody.

Connections between V and $\exists$
The two quantifiers are actually intimately connected with each other, through negation. When one says that everyone dislikes parsnips, one is also saying that there does not exist someone who likes them; and vice versa:

$\forall x \neg$*Likes(x, Parsnips)* is equivalent to $\neg \exists x$ *Likes(x, Parsnips)*

We can go one step further. "Everyone likes ice cream" means that there is no one who does not like ice cream:

$\forall$ x *Likes(x, IceCream)* is equivalent to $\neg \exists x \neg$*Likes(x, IceCream)*

Because V is really a conjunction over the universe of objects and Ǝ is a disjunction, it should not be surprising that they obey De Morgan's rules.

The De Morgan rules for quantified and unqualified sentences are as follows:

$\forall x \neg P = \neg \exists x P$ $\qquad \neg P \wedge \neg Q = \neg(P \vee Q)$
$\neg \forall x P = \exists x \neg P$ $\qquad \neg(P \wedge Q) = \neg P \vee \neg Q$
$\forall x P = \neg \exists x \neg P$ $\qquad P \wedge Q = \neg(\neg P \vee \neg Q)$
$\exists x P = \neg \forall x \neg P$ $\qquad P \vee Q = \neg(\neg P \wedge \neg Q)$

Thus, we do not really need both V and $\exists$, just as we do not really need both Λ and V. For the purposes of AI, the content, and hence the readability, of the sentences are important. Therefore, we will keep both of the quantifiers.

A new statement is true by proving that it follows from the statements that are already known.
Thus the idea of a proof, as developed in mathematics as a rigorous way of demonstrating the truth of an already believed proposition, can be extended to include deduction as a way of deriving answers to questions and solutions to problems.

So we move to first-order predicate logic as a way of representing knowledge because it permits representations of things that cannot reasonably be represented in prepositional logic.

Well-formed Formulas (wffs)

Defined recursively
- Literals are wffs
- wffs connected by ¬, □ , □, and □ are wffs
- wffs surrounded by quantifiers are wffs

In predicate logic, we can represent real-world facts as statements written as wff's.
But a major motivation for choosing to use logic

1. A good way of reasoning with that knowledge is available.

2. Determining the validity of a proposition in propositional logic is straightforward.
3. It provides a way of deducing new statements from old ones. But, it does not possess a decision procedure. There do exist procedures that will find a proof of a proposed theorem if indeed it is a theorem. But these procedures are not guaranteed to halt if the proposed statement is not a theorem. F*irst-order predicate logic is not decidable, it is semidecidable.*
4. A simple such procedure is to use the *rules of inference to generate theorem's* from the axioms in some orderly fashion, testing each to see if it is the one for which a proof is sought.

Let's now explore the use of predicate logic as a way of representing knowledge by looking at a specific example. Consider the following set of sentences:

1. Marcus was a man.
2. Marcus was a Pompeian.
3. All Pompeian were Romans.
4. Caesar was a ruler.
5. All Romans were either loyal to Caesar or hated him.
6. Everyone is loyal to someone.
7. People only try to assassinate rulers they are not loyal to.
8. Marcus tried to assassinate Caesar.

The facts described by these sentences can be represented as a set of wff's in predicate logic as follows:

1. Marcus was a man.
Man(Marcus)

This representation captures the critical fact of Marcus being a man. It fails to capture some of the information in the English sentence, namely the notion of past tense. Whether this omission is acceptable or not depends on the use to which we intend to put the knowledge. For this simple example, it will be all right.

2. Marcus was a Pompeian.
Pompeian(Marcus)
3. All Pompeians were Romans.
Vs : Pompeian(x) → Romon(x)
4. Caesar was a ruler.
Ruler(Caesar)

Here we ignore the fact that proper names are often not references to unique individuals, since many people share the same name. Sometimes deciding which of several people of the same name is being referred to in a particular statement may require a fair amount of knowledge and reasoning.

5. All Romans were either loyal to Caesar or hated him.
Vx: Roman(x) →[(loyalto(x, Caesar) V hate(x,Caesar)

In English. the word "or" sometimes means the logical inclusive-or and sometimes means the logical exclusive-or (XOR). Here we have used the inclusive interpretation. To express that, we would have to write:

$\forall x$ Roman(x) → [(loyalto(x, Caesar) V hate(x, Caesar)) ∧ ┐(loyalto(x, Caesar) ∧ hate(x, Caesar))]

6. Everyone is loyal to someone.
   $\forall x$:∃ y : →loyalto(x,y)

A major problem that arises when trying to convert English sentences into logical statements is the *scope of quantifiers*. Does this sentence say, as we have assumed in writing the logical formula above, that for each person there exists someone to whom he or she is loyal, possibly a different someone for everyone? Or does it say that there exists someone to whom everyone is loyal (which would be written as ∃y :$\forall x$: loyalto(x, y))? Often only one of the two interpretations seems likely, so people tend to favor it.

7. People only try to assassinate rulers they are not loyal to.
   $\forall x$: $\forall y$ person(x) ∧ ruler(y) ∧ tryassassinate(x, y) → ┐ loyalto(x, y)
8. Marcus tried to assassinate Caesar.
   tryassassinate (Marcus, Caesar)

From this brief attempt to convert English sentences into logical statements, it should be clear how difficult the task is. For a good description of many issues involved in this process. Now suppose that we want to use these statements to answer the question

Was Marcus loyal to Caesar?

It seems that using 7 and 8, we should be able to prove that Marcus was not loyal to caesar (again ignoring the distinction between past and present tense).

Now let's try to produce a formal proof, **reasoning backward from the desired goal:**

┐ loyalto(Marcus, Caesar)

In order to prove the goal, we need to use the rules of inference to transform it into ***another goal*** *(or possibly a set of goals) that can in turn be transformed, and so on, until there are no unsatisfied goals remaining.*

This process may require the search of an AND-OR graph when there are alternative ways of satisfying individual goals. Here, for simplicity, we show only a single path. Figure 5.2 shows an attempt to produce a **proof of the goal by reducing the set of necessary but as yet unattained goals** to the empty set. The attempt fails, however, since there is no way to satisfy the goal person (Marcus) with the statements we have available.

The problem is that, although we know that Marcus was a man, we do not have any way to conclude from that that Marcus was a person. We need to add the representation of another fact to our system, namely:

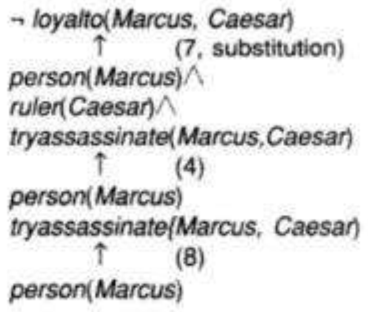


Fig. 5.2 *An Attempt to Prove ¬loyalto(Marcus,Caesar)*

8. All men are people.
   Vx : man(x)→ person(x)

Now we can satisfy the last goal and produce a proof that Marcus was not loyal to Caesar. *Three important issues must be addressed in the process of convening English sentences into logical statements and then using those statements to deduce new ones:*

1. Many English sentences are **ambiguous** (for example. 5, 6, and 7 above). Choosing the correct interpretation may be difficult.
2. There is often a choice of how to represent the knowledge (as discussed in connection with 1 , and 7 above). *Simple representations are desirable.* but they may preclude certain kinds of reasoning. The expedient representation for a particular set of sentences depends on the use to which the knowledge contained in the sentences will be put.
3. Even in very simple situations. *a set of sentences is unlikely to contain all the information necessary to reason about the topic at hand.* In order to be able to use a set of statements effectively, it is usually necessary to have access to another set of statements that represent facts that people consider too obvious to mention.

An additional problem arises in situations where we do not know in advance which statements to deduce. In the example just presented, the object was to answer the question "Was Marcus loyal to Caesar?' How would a program decide whether it should try to prove

loyalto(Marcus. Caesar)
┐ loyalto(Marcus. Caesar)

Reasoning backward from a proposed truth to the axioms can be done and instead try to reason forward and see which answer it gets to.
The problem with this forward approach is that,

1. The branching factor going forward from the axioms is so great that it would probably not get to either answer in any reasonable amount of time.
2. It could use some sort of heuristic rules for deciding which answer is more likely and then try to prove that one first. If it fails to find a proof after some reasonable amount of effort, it can try the other answer. This notion of limited effort is important, since any proof procedure we use may not halt if given a nontheorem.
3. Try to prove both answers simultaneously and stop when one effort is successful. Even here, however, if there is not enough information available to answer the question with certainty,

the program may never halt. Yet a fourth strategy is to try both to prove one answer and to disprove it, and to use information gained in one of the processes to guide the other.

REPRESENTING INSTANCE AND ISA RELATIONSHIPS

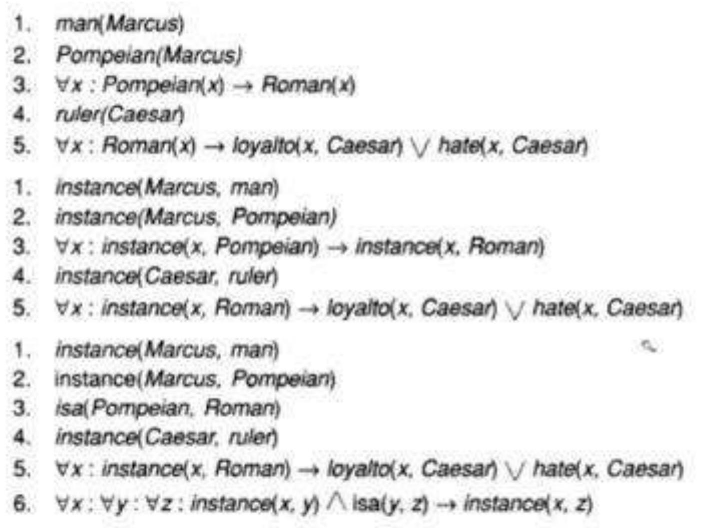
The specific attributes instance and isa are play in a particularly useful form of reasoning, property inheritance. Figure 5.3 shows the first five sentences of the last section represented in logic in three different ways.

The first part of the figure contains the representations we have already discussed. In these representations, class membership is represented with unary predicates (such as Roman), each of which corresponds to a class. Asserting that P(x) is true is equivalent to asserting that x is an instance (or element) of P

The second part of the figure contains representations that use the instance predicate explicitly

The third part contains representations that use both the instance and isa predicates explicitly. The use of the isa predicate simplifies the representation of sentence 3, but it requires that one additional axiom (shown here as number 6) be provided.

This additional axiom describes how an instance relation and an isa relation can be combined to derive a new instance relation. This one additional axiom is general, though, and does not need to be provided separately for additional isa relations.



1. $man(Marcus)$
2. $Pompeian(Marcus)$
3. $\forall x : Pompeian(x) \rightarrow Roman(x)$
4. $ruler(Caesar)$
5. $\forall x : Roman(x) \rightarrow loyalto(x, Caesar) \vee hate(x, Caesar)$

1. $instance(Marcus, man)$
2. $instance(Marcus, Pompeian)$
3. $\forall x : instance(x, Pompeian) \rightarrow instance(x, Roman)$
4. $instance(Caesar, ruler)$
5. $\forall x : instance(x, Roman) \rightarrow loyalto(x, Caesar) \vee hate(x, Caesar)$

1. $instance(Marcus, man)$
2. $instance(Marcus, Pompeian)$
3. $isa(Pompeian, Roman)$
4. $instance(Caesar, ruler)$
5. $\forall x : instance(x, Roman) \rightarrow loyalto(x, Caesar) \vee hate(x, Caesar)$
6. $\forall x : \forall y : \forall z : instance(x, y) \wedge isa(y, z) \rightarrow instance(x, z)$

Fig. 5.3 *Three Ways of Representing Class Membership*

Pompeian(Paulus)

┐ [loyalto(Paulus, Caesar) V hate(Paulets.Caesar)]

For convenience, we now return to our original notation using unary predicates to denote class relations.

In other words, suppose we want to make Paulus an exception to the general rule about Romans and their feelings toward Caesar. Unfortunately, we cannot simply add these facts to our existing knowledge base the way we could just add new nodes into a semantic net. The difficulty is that if the old assertions are left unchanged, then the addition of the new assertions makes the knowledge base inconsistent. In order to restore consistency, it is necessary to modify the original assertion to which an exception is being made. So our original sentence 5 must become:

$$V\ x : Roman(x) \wedge \neg\ eq(x, Paulus) \rightarrow loyalto(x, Caesar)\ V\ hate(x, Caesar)$$

In this framework, every exception to a general rule' must be stated twice, once in a particular statement and once in an exception list that forms part of the general rule. This makes the use of general rules in this framework less convenient and less efficient when there arc exceptions than is the use of general rules in a semantic net. A further problem arises when information is incomplete and it is not possible to prove that no exceptions apply in a particular instance.

USING FIRST-ORDER LOGIC (KNOWLEDGE REPRESENTATION IN FOL)

In knowledge representation, a **domain** is a section of the world about which we wish to express some knowledge.

Assertions and queries in first-order logic

Sentences are added to a knowledge base using TELL, exactly as in propositional logic. Such sentences are called **assertions**.

For example, we can assert that John is a king and those kings are persons:
TELL(KB, King(John))
TELL(KB, ∀ x King(x) => Person(x))
We can ask questions of the knowledge base using ASK. For example
ASK(KB, King(John))

Return true. Questions asked using ASK are called queries or goals. Generally speaking, any query that is logically entailed by the knowledge base should be answered affirmatively. For example, given the two assertions in the preceding paragraph, the query
ASK(KB, Person(John))
Should also return true. We can also ask quantified queries, such as
ASK(KB, ∃ x Person(x))
The answer to this query could be true, but this is neither helpful nor amusing. A query with existential variables is asking "Is there an x such that …" and we solve it by providing such an x.

The standard form for an answer of this sort is a substitution or binding list, which is a set of variable/ term pair.

In this particular case, given just the two assertions, the answer would be {x/John}. If there is more than one possible answer, a list of substitutions can be returned.

Axioms, definitions, and theorems

Axioms are facts and rules that attempt to capture all of the (important) facts and concepts about a domain; axioms can be used to prove theorems

- Each of these sentences is an axiom, as they provide basic factual information
- They are definitions since they have the form:
    - $\forall x,y\ P(x,y)$ □ $\Leftrightarrow \ldots$
- Some logical sentences are theorems
    - $\forall x,y\ Sibling(x,y) \Leftrightarrow Sibling(y,x)$
- Theorems provide no new information, as they are entailed by the axioms
- But they reduce the computational cost of deriving new sentences

From a purely logical point of view,
- A knowledge base contains only axioms and no theorems, because the theorems do not increase the set of conclusions that follow from the knowledge base.
- Theorems are essential to reduce the computational cost of deriving new sentences.

Without them, a reasoning system has to start from first principles every time, rather like a physicist having to rederive the rules of calculus for every new problem.

Not all axioms are definitions. Some provide *more general information* about certain predicates without constituting a definition. Indeed, some predicates *have no complete definition becaus*e we do not know enough to characterize them fully.

For example, there is no obvious way to complete the sentence:
$\forall$ *x Person(x)* <=> ...
Fortunately, first-order logic allows us to make use of the *Person* predicate without completely defining it. Instead we can write partial specifications of properties that every person has and properties that make something a person:
$\forall$ x *Person(x)* => ...
$\forall$ x ... => *Person(x)*
Axioms can also be "just plain facts", such as Male(Jim) and Spouse(Jim,Laura). Such facts form the descriptions of specific problem instances, enabling specific questions to be answered. The answers to these questions will then be theorems that follow from the axioms.

An axiom of the form $\forall$ *x,y P(x,y)* = ... is often called a **definition** of *P*, because it serves to define exactly for what objects *P* does and does not hold. It is possible to have several definitions for the same predicate;

Numbers, sets and lists
We want to be able to represent individual sets, including the empty set. We need a way to build up sets by adding an element to a set or taking the union or intersection of two sets. We will want to know if an element is a member of a set, and to be able to distinguish sets from objects that are not sets.

We will use the normal vocabulary of set theory: *EmptySet* is a constant written as {}. There is only unary predicate, Set, which is true of sets. The binary predicates are x ϵ s and s1 s2. The binary functions are s1 s2, s1 s2 , and {x/ s}

The following eight axioms provide this:
1. The only sets are the *empty set and those made by adjoining something to a set*:
∀s set(s) <=> (s=EmptySet) v (∃x,r Set(r) ^ s=Adjoin(s,r))

2. *The empty set has no elements adjoined to it:*
~ ∃x,s Adjoin(x,s)=EmptySet

3. *Adjoining an element already in the set has no effect*:
∀x,s Member(x,s) <=> s=Adjoin(x,s)

4. The only *members of a set are the elements that were adjoined into it*:
∀x,s Member(x,s) <=> ∃y,r (s=Adjoin(y,r) ^ (x=y ∨ Member(x,r)))

5. A *set is a subset of another iff all of the 1st set's members are members of the* 2nd:
∀s,r Subset(s,r) <=> (∀x Member(x,s) => Member(x,r))

6. *Two sets are equal iff each is a subset of the other*:
∀s,r (s=r) <=> (subset(s,r) ^ subset(r,s))

7. *Intersection*
∀x,s1,s2 member(X,intersection(S1,S2)) <=> member(X,s1) ^ member(X,s2)

8. *Union*
∃x,s1,s2 member(X,union(s1,s2)) <=> member(X,s1) ∨ member(X,s2)

lists
The domain of lists is very similar to the domain of sets. The difference is that lists are ordered, and the ***same element can appear more than once in a list****.*

We can use the vocabulary of Lisp for lists:
Nil is the constant list with no elements;
Cons, Append, First, and *Rest* are functions; and
Find is the predicate that does for lists what *Member* does for sets.
List? is a predicate that is true only of lists.

As with sets, it is common to use syntactic sugar in logical sentences involving lists. The empty list is [].
The term Cons(x,y), where y is a nonempty list, is written [x | y].
The term Cons(x,Nil) is written as [x].

A list with several elements, such as [A,B,C], corresponds to the nested term Cons(A,Cons(B,Cons(C,Nil)))

These rules exhibit a trivial form of the reasoning process called **perception.**

Classification of percept sentences:

1. The *sentences dealing with time have been* ***synchronic*** ("same time") rules, because they relate properties of a world state to other properties of the same world state.
2. The sentences that allow *reasoning "across time" are called* ***diachronic*****; for** example, the agent needs to know how to combine information about its previous location with information about the action just taken in order to determine its current location.

There are two kinds of synchronic rules that could allow such deductions:

Diagnostic rules:

Diagnostic rules infer the presence of hidden properties directly from the percept-derived information. For finding pits, the obvious diagnostic rules say that if a square is breezy, some adjacent square must contain a pit, or

∀ *s Breezy(s)* => ∃ r Adjacent(r,s) Λ Pit*(r)*

And that if a square is not breezy, no adjacent square contains a pit:

∀ s ¬Breezy(s) *)* => ¬ ∃ r Adjacent(r,s) Λ Pit*(r)*

Combining these two, we obtain the biconditional sentence

∀ *s Breezy(s)* <=> ∃ r Adjacent(r,s) Λ Pit*(r)*

Causal rules:

Causal rules **reflect the assumed direction of causality in the world**: some hidden property of the world causes certain percepts to be generated. For example, we might have rules stating that squares adjacent to wumpuses are smelly and squares adjacent to pits are breezy:

∀r Pit(r) => [∀s Adjacent(r,s) => Breezy(s)]

And if all squares adjacent to a given square are pitless, the square will not be breezy:

∀s [Vr *Adjacent(r,s) => ¬Pit(r)] => ¬Breezy(s)]*

Systems that reason with causal rules are called **model-based reasoning** systems, because the casual rules form a model of how the environment operates. The distinction between model-based and diagnostic reasoning is important in many areas of AI.

5. PROGLOG Programming:-

- A system in which KB can be constructed and used.
- A relation between logic and algorithm is summed up in Robert Kowalsh equation

  Algorithm = Logic + Control
- Logic programming languages, usually use backward chaining and input/output of programming languages.
- A logic programming language makes it possible to write algorithms by augmenting logic sentences with information to control the inference process.
- For Example:- PROLOG
  - ✓ A prolog program is described as a series of logical assertions each of which is a Horn Clause.
  - ✓ A Horn Clause is a Clause that has atmost one positive literal,

    Example: - P, $\neg P \wedge Q$
  - ✓ Implementation: - All inferences are done by backward chaining, with depth first search. The order of search through the conjuncts of an antecedent is left to right and the clauses in the KB are applied first-to- last order.
- **Example for FOL to PROLOG conversion:-**
  - **FOL**
    - ✓ **$\forall x$ Pet(x) $\wedge$ Small(x) $\Rightarrow$ Apartment(x)**
    - ✓ **$\forall x$ Cat(x) v Dog(x) $\Rightarrow$ Pet(x)**
    - ✓ **$\forall x$ Product(x) $\Rightarrow$ Dog(x) $\wedge$ Small(x)**
    - ✓ **Poodle(fluffy)**
  - **Equivalent PROLOG representation**
    - ✓ **Apartment(x) :- Pet(x), Small(x)**
    - ✓ **Pet(x) :- Cat(x)**

      Pet(x) :- Dog(x)
    - ✓ **Dog(x) :- Poodle(x)**

      Small(x) :- Poodle(x)
    - ✓ **Poodle(fluffy)**
  - In the PROLOG representation the consequent or the left hand side is called as head and the antecedent or the right hand side is called as **body.**
- **Execution of a PROLOG program:-**
  - The execution of a prolog program can happen in two modes,
    1. Interpreters
    2. Compilers
  - **Interpretation:**
    - ✓ A method which uses **BACK-CHAIN** algorithm with the program as the **KB.**
    - ✓ To maximize the speed of execution, we will consider two different types of constraints executed in sequence, They are
      1. **Choice Point: -** Generating sub goals one by one to perform interpretation.
      2. **Trail: -** Keeping track of all the variables that have been bound in a stack is called as trail.
  - **Compilation:-**

✓ Procedure implementation is done to run the program (i.e.) calling the inference rules whenever it is required for execution

6. UNIFICATION

Lifted inference rules require finding substitutions that make different logical expressions look identical. This process is called unification and is a key component of all first-order inference algorithms.

- The process of finding all legal substitutions that make logical expressions look identical and
- Unification is a "pattern matching" procedure that takes two atomic sentences, called literals, as input, and returns "failure" if they do not match and a substitution list, Theta, if they do match.

The Unify algorithm takes two sentences and returns a unifier for them if one exists:

UNIFY (p, q) = q where SUBST (θ, p) = SUBST (θ, q).

Let us look at some examples of how Unify should behave. Suppose we have a query Knows (John, x): whom does John know? Some answers to this query can be found by finding all sentences in the knowledge base that unify with Knows (John, x).

Here are the results of unification with four different sentences that might be in the knowledge base.

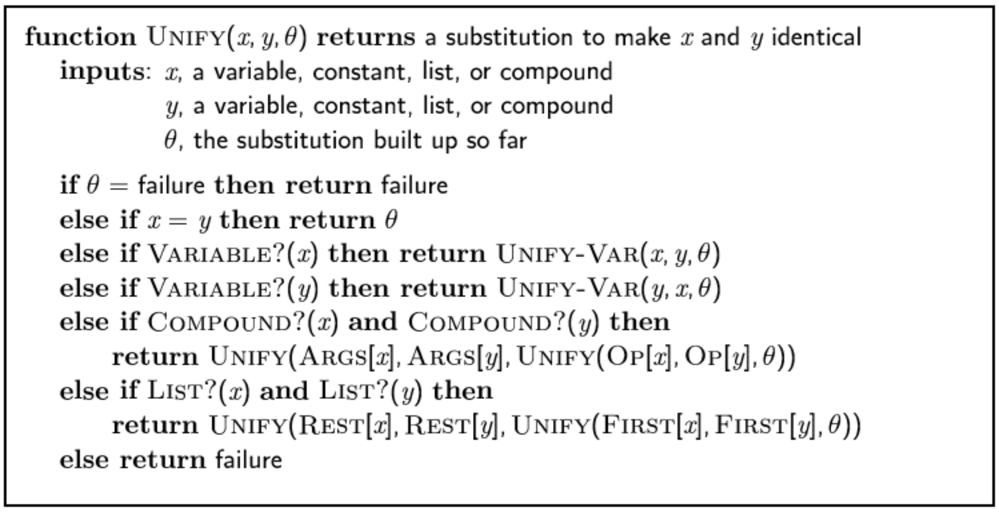
UNIFY (Knows (John, x), Knows (John, Jane)) = {x/Jane}
UNIFY (Knows (John,x), Knows (y, Bill))= {x/Bill, y/John}
UNlFY (Knows (John,x), Knows (y, Mother(y))) = {y/John, x/Mother(John)}
UNlFY (Knows (John,x), Knows (x, Elizabeth)) = fail .

The last unification fails because x cannot take on the values John and Elizabeth at the same time. Now, remember that Knows(x, Elizabeth) means "Everyone knows Elizabeth," so we should be able to infer that John knows Elizabeth.

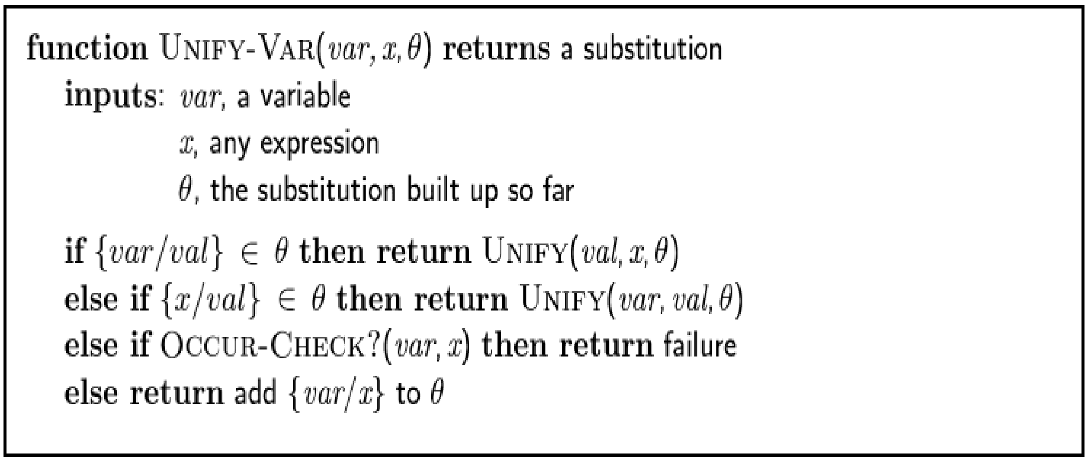
The problem arises only because the two sentences happen to use the same variable name, x. The problem can be avoided by standardizing apart one of the two sentences being unified, which means renaming its variables to avoid name clashes.

This so-called **occur check** makes the complexity of the entire algorithm quadratic in the size of the expressions being unified. Some systems, including all logic programming systems, simply omit the occur check and sometimes make unsound inferences as a result; other systems use more complex algorithms with linear-time complexity.

The Unification algorithm



```
function UNIFY($x, y, \theta$) returns a substitution to make $x$ and $y$ identical
   inputs: $x$, a variable, constant, list, or compound
           $y$, a variable, constant, list, or compound
           $\theta$, the substitution built up so far

   if $\theta$ = failure then return failure
   else if $x = y$ then return $\theta$
   else if VARIABLE?($x$) then return UNIFY-VAR($x, y, \theta$)
   else if VARIABLE?($y$) then return UNIFY-VAR($y, x, \theta$)
   else if COMPOUND?($x$) and COMPOUND?($y$) then
       return UNIFY(ARGS[$x$], ARGS[$y$], UNIFY(OP[$x$], OP[$y$], $\theta$))
   else if LIST?($x$) and LIST?($y$) then
       return UNIFY(REST[$x$], REST[$y$], UNIFY(FIRST[$x$], FIRST[$y$], $\theta$))
   else return failure
```



```
function UNIFY-VAR($var, x, \theta$) returns a substitution
   inputs: $var$, a variable
           $x$, any expression
           $\theta$, the substitution built up so far

   if $\{var/val\} \in \theta$ then return UNIFY($val, x, \theta$)
   else if $\{x/val\} \in \theta$ then return UNIFY($var, val, \theta$)
   else if OCCUR-CHECK?($var, x$) then return failure
   else return add $\{var/x\}$ to $\theta$
```

7. **FORWARD CHAINING**

Using a deduction to reach a conclusion from a set of antecedents is called forward chaining. In other words,the system starts from a set of facts, and a set of rules, and tries to find the way of using these rules and facts to deduce a conclusion or come up with a suitable cause of action. This is known as data driven reasoning.

Steps for forward-chaining algorithm

The first forward chaining algorithm,

1. Starting from the known facts
2. Triggers all the rules whose premises are satisfied
3. Add their conclusions to the known facts.
4. The process repeats until the query is answered (assuming that just one answer is required) or no new facts are added.

Note:

(a) The initial facts appear in the bottom level
(b) Facts inferred on the first iteration is in the middle level
(c) The facts inferered on the 2nd iteration is at the top level

```
function FOL-FC-ASK($KB, \alpha$) returns a substitution or $false$
    repeat until $new$ is empty
        $new \leftarrow \{\}$
        for each sentence $r$ in $KB$ do
            $(p_1 \wedge \ldots \wedge p_n \Rightarrow q) \leftarrow$ STANDARDIZE-APART($r$)
            for each $\theta$ such that $(p_1 \wedge \ldots \wedge p_n)\theta = (p'_1 \wedge \ldots \wedge p'_n)\theta$
                        for some $p'_1, \ldots, p'_n$ in $KB$
                $q' \leftarrow$ SUBST($\theta, q$)
                if $q'$ is not a renaming of a sentence already in $KB$ or $new$ then do
                    add $q'$ to $new$
                    $\phi \leftarrow$ UNIFY($q', \alpha$)
                    if $\phi$ is not $fail$ then return $\phi$
        add $new$ to $KB$
    return $false$
```

Forward Chaining algorithm

EXAMPLE

We will prove that West is a criminal. First, we will represent these facts as first-order definite clauses. "... **it is a crime for an American to sell weapons to hostile nations**":

American(x) A Weapon(y) A Sells(x,y,z) A Hostile(z) $\Rightarrow$ Criminal(x) . (3.3)

"Nono ... has some missiles." The sentence $\exists$x Owns(Nono, x) Λ Missile(x) is transformed into two definite clauses by Existential Elimination, introducing a new constant M1

Owns (Nono, M1) (3.4)

Missile (M1) (3.5)

"All of its missiles were sold to it by Colonel West":

Missile(x) A Owns (Nono, x) => Sells (West, x, Nono) . (3.6)

We will also need to know that missiles are weapons:

Missile(x) => Weapon(x) (3.7)

and we must know that an enemy of America counts as "hostile":

Enemy (x, America) => Hostile(x). (3.8)

"West, who is American ...":

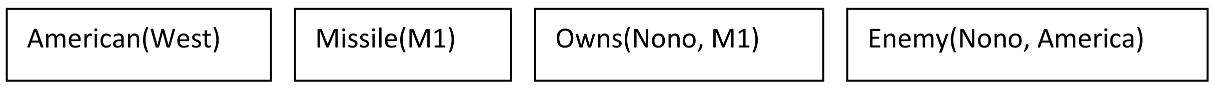
American (West) . (3.9)

"The country Nono, an enemy of America ...":

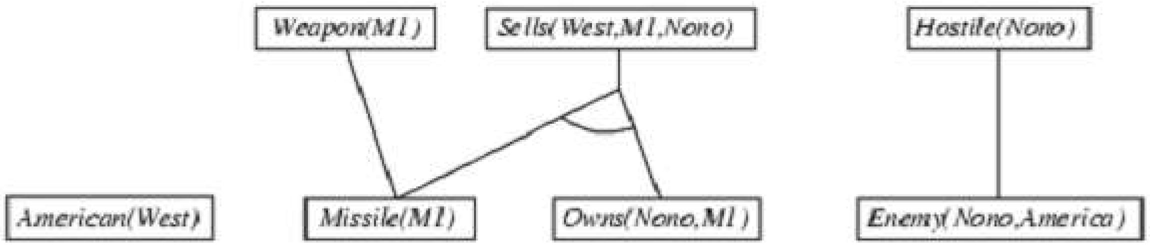
Enemy (Nono, America) . (3.10)

Figure: The proof tree generated by forward chaining on the crime example

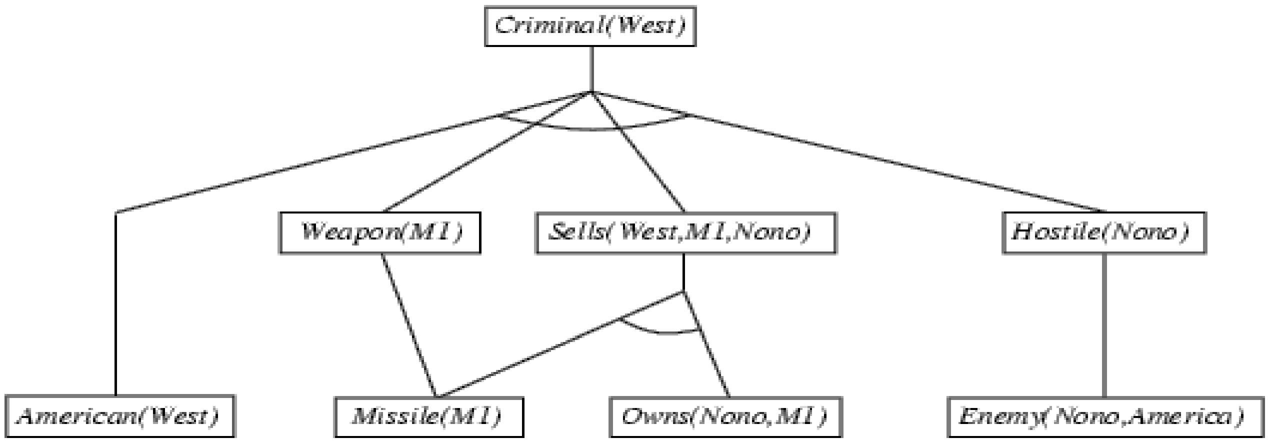
Step 1:



Step 2:



Step 3:



Forward chaining applies a set of rules and facts to deduce whatever conclusions can be derived.

FC Algorithm has three possible types of complexity

- ✓ **Pattern Matching: -** "inner loop" of the algorithm involves finding all possible unifiers such that the premise of a rule unifies with a suitable set of facts in the **KB.**
- ✓ **Matching rules against known facts:-** The algorithm re-checks every rule on every iteration to see whether its premises are satisfied, even if very few additions are made to the **KB** on each iteration
- ✓ Irrelevant facts to the goal are generated

8. BACKWARD CHAINING

In **backward chaining**, we start from a **conclusion**, which is the hypothesis we wish to prove and we aim to show how that conclusion can be reached from the rules and facts in the data base.The conclusion we are aiming to prove is called a goal and the reasoning in this way is known as **goal-driven**.

Note:

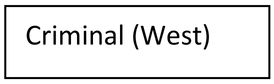
(a) To prove Criminal (West), we have to prove four conjuncts below it.

(b) Some of which are in knowledge base, and others require further backward chaining.

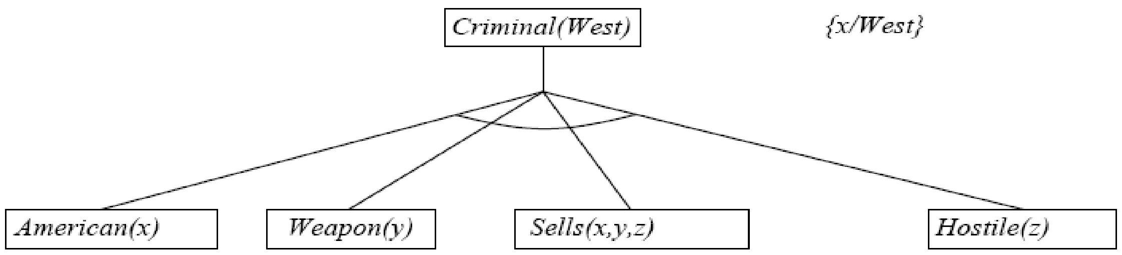
```
function FOL-BC-ASK($KB$, goals, $\theta$) returns a set of substitutions
    inputs: $KB$, a knowledge base
            $goals$, a list of conjuncts forming a query
            $\theta$, the current substitution, initially the empty substitution { }
    local variables: $ans$, a set of substitutions, initially empty

    if $goals$ is empty then return $\{\theta\}$
    $q' \leftarrow$ SUBST($\theta$, FIRST($goals$))
    for each $r$ in $KB$ where STANDARDIZE-APART($r$) = $(p_1 \wedge \ldots \wedge p_n \Rightarrow q)$
            and $\theta' \leftarrow$ UNIFY($q, q'$) succeeds
      $ans \leftarrow$ FOL-BC-ASK($KB$, $[p_1, \ldots, p_n|$REST($goals$)], COMPOSE($\theta, \theta'$)) $\cup$ $ans$
    return $ans$
```

<u>Proof tree constructed by backward chaining to prove that West is a criminal.</u>

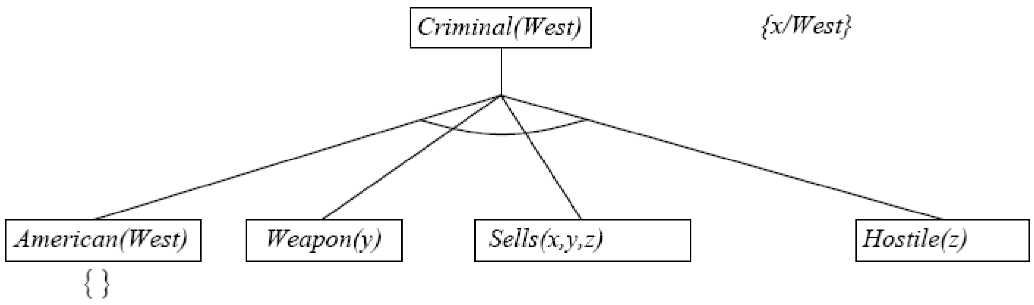
Step 1:



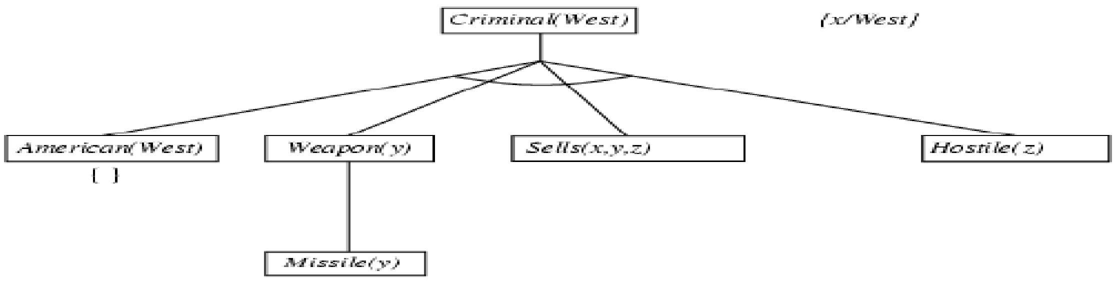
Step2:



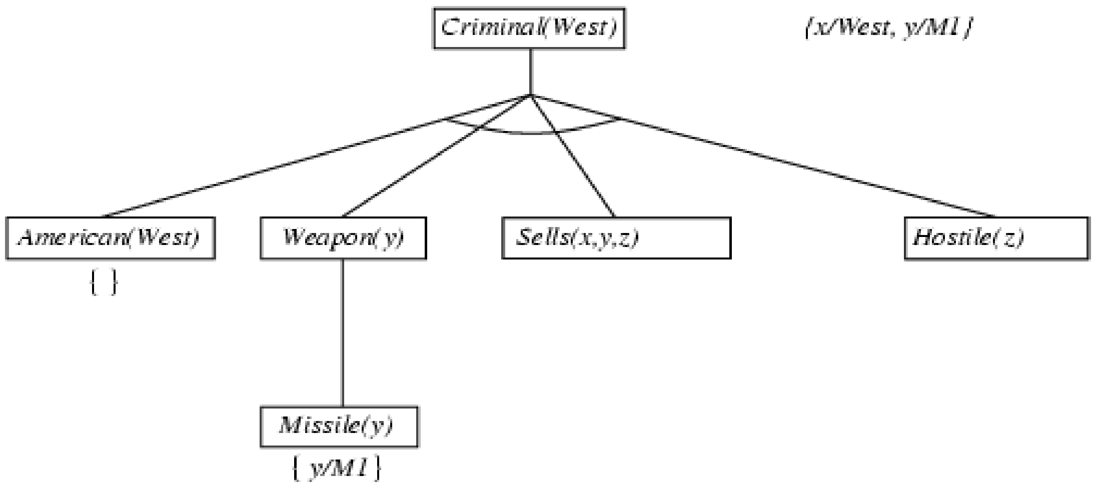
Step 3



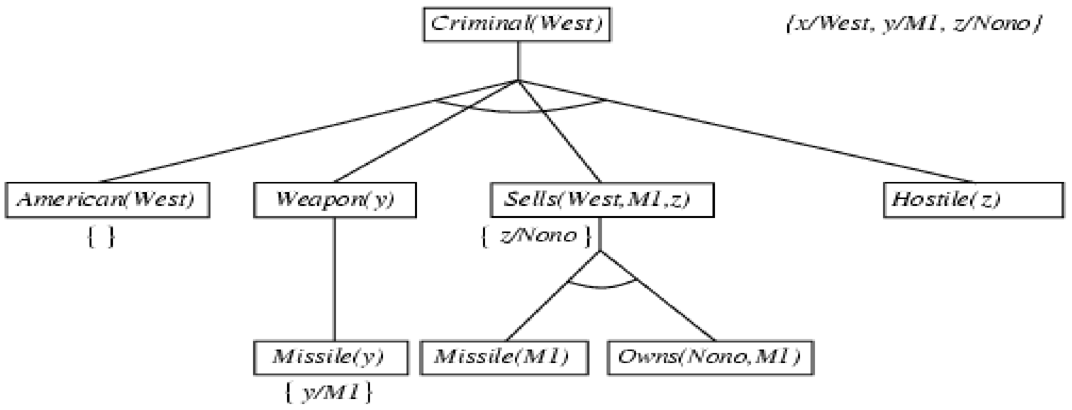
Step 4:



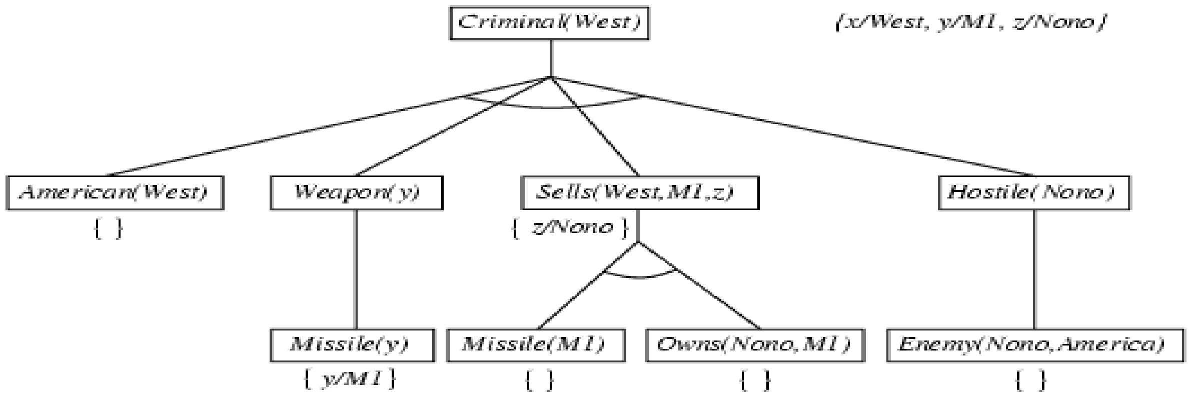
Step 5:



Step 6:



Step 7:



The tree should be read depth first, left to right.

- To prove Criminal (West), we have to prove the four conjuncts below it. Some of these are in the knowledge base, and others require further backward chaining.
- Bindings for each successful unification are shown next to the corresponding subgoal. Note that once one subgoal in a conjunction succeeds, its substitution is applied to subsequent subgoals.
- Thus, by the time FOL-BC-ASK gets to the last conjunct, originally Hostile(z), z is already bound to Nono.

- The algorithm uses **composition** of substitutions. COMPOSE (θ_1, θ_2) is the substitution whose effect is identical to the effect of applying each substitution in turn. That is,

SUBST (COMPOSE (θ_1, θ_2),p) = SUBST (θ_2,SUBST (θ_1,p)) .

In the algorithm, the current variable bindings, which are stored in θ, are composed with the bindings resulting from unifying the goal with the clause head, giving a new set of current bindings for the recursive call.

9. RESOLUTION

- Resolution yields a complete inference algorithm when coupled with any complete search algorithm.
- Resolution makes use of the inference rules.
- Resolution performs deductive inference.
- Resolution uses proof by contradiction.
- One can perform Resolution from a Knowledge Base. A Knowledge Base is a collection of facts or one can even call it a database with all facts.

In 1930, the German mathematician Kurt Godel proved the first **completeness theorem** for first-order logic, showing that any entailed sentence has a finite proof.

In 1931, Godel proved an even more famous **incompleteness theorem**. The theorem states that a logical system that includes the principle of induction—without which very little of discrete mathematics can be constructed—is necessarily incomplete. Hence, there are sentences that are entailed, but have no finite proof within the system.

Steps for Resolution

1. Convert all facts to clause form (product of sums)
2. Negate the goal (theorem to be proven)
3. Convert the negated goal to clause form, and add to the set of facts
4. Until NIL is found, repeat
    - Select two clauses
    - Compute their resolvent
    - Add it to the set of facts

Conjunctive normal form for first-order logic

First-order resolution requires that sentences be in conjunctive normal form (CNF)—that is**, a conjunction of clauses, where each clause is a disjunction of literals. Literals can contain variables, which are assumed to be universally quantified**. For example, the sentence

$\forall x$ American (x) Λ Weapon (y) Λ Sells (x, y, z) Λ Hostile (z) =>Criminal (x)

becomes, in CNF,

¬American (x) V ¬Weapon (y) V ¬Sells (x, y, z) V ¬Hostile (z) V Criminal (x) .

A clause can also be represented as an implication with a conjunction of atoms on the left and a disjunction of atoms on the right. This form, sometimes called Kowalski form when written with a right-to-left implication symbol, is often much easier to read.
Every sentence of first-order logic can be converted into an inferentially equivalent CNF sentence. In particular, the CNF sentence will be unsatisfiable just when the original sentence is unsatisfiable, so we have a basis for doing proofs by contradiction on the CNF sentences.

CNF CONVERSION
The steps are as follows:

Step 1: Eliminate implications:

$\forall x\ [\neg\forall y\ \neg Animal(y) \lor Loves(x,y)] \lor [\exists y\ Loves(y,x)]$.

Step 2: Move ¬ inwards:

In addition to the usual rules for negated connectives, we need rules for negated quantifiers. Thus, we have

$\neg\forall x\ p$ becomes $\exists x\ \neg p$

$\neg\exists x\ p$ becomes $\forall x\ \neg p$.

Our sentence goes through the following transformations:

$\forall x\ [\exists y\ \neg\ (\neg Animal(y) \lor Loves(x,y))] \lor [\exists y\ Loves(y,x)\}$.

$\forall x\ [\exists y\ \neg\neg Animal(y) \land \neg Loves(x,y)] \lor [\exists y\ Loves(y,x)]$.

$\forall x\ [\exists y\ Animal(y) \land \neg Loves(x,y)] \lor [\exists y\ Loves(y,x)]$.

Notice how a universal quantifier ($\forall$ y) in the premise of the implication has become an existential quantifier. The sentence now reads "Either there is some animal that x doesn't love, or (if this is not the case) someone loves x." Clearly, the meaning of the original sentence has been preserved.

Step 3: Standardize variables:

For sentences like $(\forall x\ P(x)) \lor (\exists x\ Q(x))$ which use the same variable name twice, change the name of one of the variables. This avoids confusion later when we drop the quantifiers. Thus, we have

$\forall x\ [\exists y\ Animal(y) \land \neg Loves(x,y)] \lor [\exists z\ Loves(z,x)]$.

Step 4: Skolemize:

Skolemization is the process of removing existential quantifiers by elimination. In the simple case, it is just like the Existential Instantiation rule. Translate $\exists x\ P(x)$ into $P(A)$, where A is a new constant. If we apply this rule to our sample sentence, however, we obtain

$\forall x\ [Animal(A) \land \neg Loves(x, A)] \lor Loves(B, x)$

Which has the wrong meaning entirely: it says that everyone either fails to love a particular animal A or is loved by some particular entity B. In fact, our original sentence allows each person to fail to love a different animal or to be loved by a different person. Thus, we want the Skolem entities to depend on x:

$\forall x\ [Animal(F(x)) \land \neg Loves(x, F(x))] \lor Loves(G(x), x)$.

Here F and G are **Skolem functions**. The general rule is that the arguments of Skolem function are all the universally quantified variables in whose scope the existential quantifier appears. As with Existential Instantiation, the Skolemized sentence is satisfiable exactly when the original sentence is satisfiable.

Step 5: Drop universal quantifiers:
At this point, all remaining variables must be universally quantified. Moreover, the sentence is equivalent to one in which all the universal quantifiers have been moved to the left. We can therefore drop the universal quantifiers:

$[Animal(F(x)) \wedge \neg Loves(x, F(x))] \vee Loves(G(x),x)$.

Step 6: Distribute V over Λ:

$[Animal(F(x)) \vee Loves(G(x),x)] \wedge [\neg Loves(x, F(x)) \vee Loves(G(x),x)]$.

This step may also require flattening out nested conjunctions and disjunctions.The sentence is now in CNF and consists of two clauses. It is quite unreadable. (It may help to explain that the Skolem function F(x) refers to the animal potentially unloved by x, whereas G(x) refers to someone who might love x.) Fortunately, humans seldom need look at CNF sentences—the translation process is easily automated.

The rule we have just given is the binary **resolution rule**, because it resolves exactly two literals. The binary resolution rule by itself does not yield a complete inference procedure. The full resolution rule resolves subsets of literals in each clause that are unifiable.

An alternative approach is to extend **factoring**—the removal of redundant literals—to the first-order case.

Propositional factoring reduces two literals to one if they are identical; first-order factoring reduces two literals to one if they are unifiable.

The unifier must be applied to the entire clause. The combination of binary resolution and factoring is complete.

Example proofs

Consider the following problem:

We will prove that West is a criminal. First, we will represent these facts as first-order definite clauses.

- ✓It is a crime for an American to sell weapons to hostile nations:
  ∀x y z American(x) ∧ Weapon(y) ∧ Nation(z) ∧ Hostile(z) ∧ Sells(x, y, z ⇒ Criminal(x)
- ✓ Nono … has some missiles,
  ∃x Owns(Nono,x) ∧ Missile(x)
- ✓ all of its missiles were sold to it by Colonel West

∀x Missiles(x) ∧ Owns(Nono,x) ⇒ Sells(West,x,Nono)

- ✓ We will also need to know that missiles are weapons

  ∀x Missile(x) ⇒ Weapon(x)
- ✓ An enemy of America counts as "hostile“

  ∀x Enemy(x,America) ⇒ Hostile(x)
- ✓ West, who is American …

  American(West)
- ✓ Nono, is a nation

  Nation (Nono)
- ✓ Nono, an enemy of America

  Enemy (Nono, America)
- ✓ America is nation

  Nation (America)

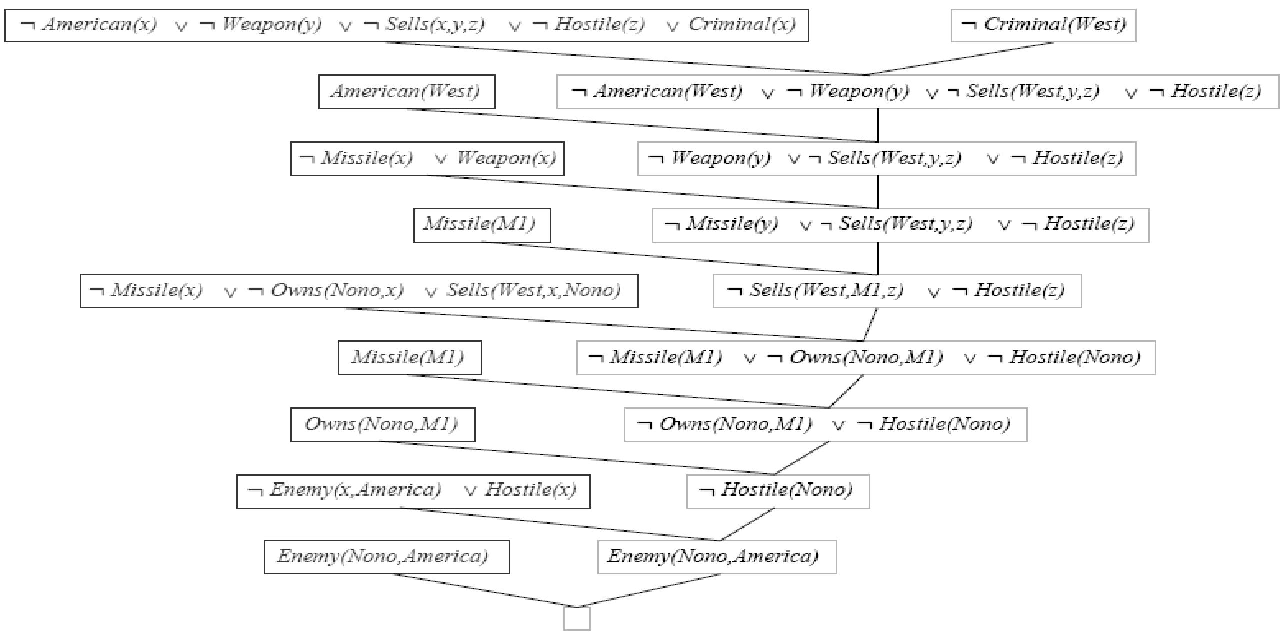
This knowledge base contains no function symbols and is therefore an instance of the class of **Datalog knowledge bases**—that is, sets of first-order definite clauses with no function symbols. The absence of function symbols makes inference much easier.

Resolution proves that KB |= a by proving KB Λ ¬α unsatisfiable, i.e., by deriving the empty clause.

¬American(x) V ¬Weapon(y) V ¬Sells(x, y, z) V ¬Hostile (z) V Criminal(x)
¬Missile(x) V ¬Owns (Nono, x) V Sells (West, x, Nono)
¬Enemy(x, America) V Hostile(x)
¬Missile(x) V Weapon(X)

American (West)
Missile (M1)
Owns (Nono, M1)
Enemy (Nono,America)

Figure : A resolution proof that West is a criminal.



¬ American(x) ∨ ¬ Weapon(y) ∨ ¬ Sells(x,y,z) ∨ ¬ Hostile(z) ∨ Criminal(x)
¬ Criminal(West)
American(West)
¬ American(West) ∨ ¬ Weapon(y) ∨ ¬ Sells(West,y,z) ∨ ¬ Hostile(z)
¬ Missile(x) ∨ Weapon(x)
¬ Weapon(y) ∨ ¬ Sells(West,y,z) ∨ ¬ Hostile(z)
Missile(M1)
¬ Missile(y) ∨ ¬ Sells(West,y,z) ∨ ¬ Hostile(z)
¬ Missile(x) ∨ ¬ Owns(Nono,x) ∨ Sells(West,x,Nono)
¬ Sells(West,M1,z) ∨ ¬ Hostile(z)
Missile(M1)
¬ Missile(M1) ∨ ¬ Owns(Nono,M1) ∨ ¬ Hostile(Nono)
Owns(Nono,M1)
¬ Owns(Nono,M1) ∨ ¬ Hostile(Nono)
¬ Enemy(x,America) ∨ Hostile(x)
¬ Hostile(Nono)
Enemy(Nono,America)
Enemy(Nono,America)

We also include the negated goal ¬Criminal (West).
Notice the structure: single "spine" beginning with the goal clause, resolving against clauses from the knowledge base until the empty clause is generated.
This is characteristic of resolution on Horn clause knowledge bases. In fact, the clauses along the main spine correspond exactly to the consecutive values of the goals variable in the backward chaining algorithm

10. KNOWLEDGE ENGINEERING IN FIRST-ORDER LOGIC

The knowledge engineer must understand enough about the domain in question to represent the important objects and relationships. He or she must also understand enough about the representation language to correctly encode these facts.

The Knowledge Engineering process

1. ***Identify the task:***

The knowledge engineer must delineate the range of questions that the knowledge base will support and the kinds of facts that will be available for each specific problem instance.

The task will determine what knowledge must be represented in order to connect problem instances to answers.

2 Assemble the relevant knowledge:

The knowledge engineer might already be an expert in the domain, or might need to work with real experts to extract what they know – a process called knowledge acquisition. At this stage, the knowledge is not represented formally. The idea is to understand the scope of the knowledge base, as determined by the task, and to understand how the domain actually works.

For real domains, the issue of relevance can be quite difficult – for example, a system for simulating VLSI designs might or might not need to take into account stray capacitances and skin effects.

3 Decide on a vocabulary of predicates, functions, and constants:

Translate the important domain-level concepts into logic-level names. This involves many choices, some arbitrary and some important.
Once the choices have been made, the result is a vocabulary that is known as the **ontology** of the domain. **The word ontology means a particular theory of the nature of being or existence. The ontology determines what kinds of things exist, but does not determine their specific properties and interrelationships.**

4 Encode general knowledge about the domain.

The knowledge engineer writes down the axioms for all the vocabulary terms. This pins down the meaning of the terms, enabling the expert to check the content.

Often, this step reveals misconceptions or gaps in the vocabulary that must be fixed by returning to step 3 and iterating through the process.

5. Encode a description of the specific problem instance.

If the ontology is well thought out, this step will be easy. It will mostly involve writing simple atomic sentences about instances of concepts that are already part of the ontology.

For the logical agent, problem instances are supplied by the sensors, whereas a “disembodied” knowledge base is supplied with additional sentences in the same way that traditional programs are supplied with input data.

6. Pose queries to the inference procedure and get answers.

This is where the reward is: we can let the inference procedure operate on the axioms and problem-specific facts to derive the facts we are interested in knowing.

7. Debug the knowledge base:

The answers to queries will seldom be correct on the first try. More precisely, the answers will be correct for the knowledge base as written, assuming that the inference procedure is sound, but they will not be the ones that the user is expecting.

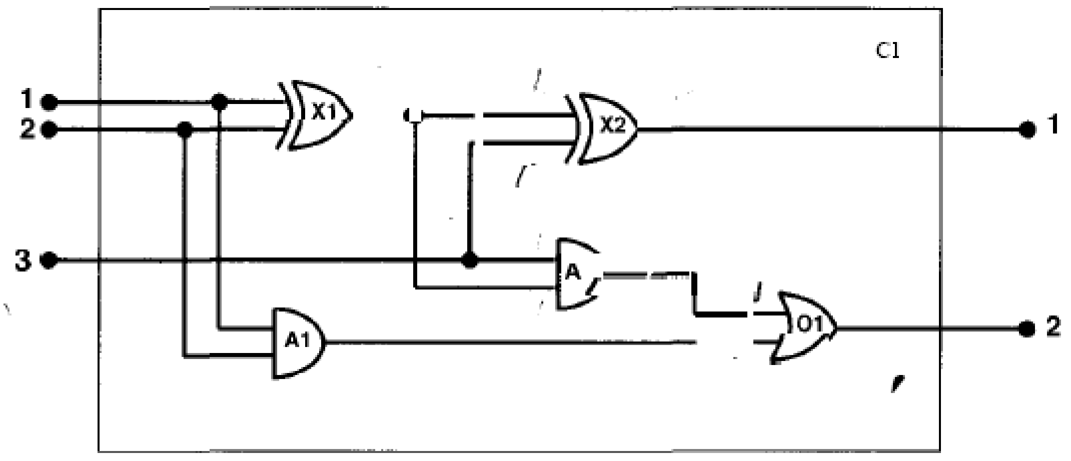
For example, if an axiom is missing, some queries will not be answerable from the knowledge base. A considerable debugging process could ensue.

Missing axioms or axioms that are too weak can be identified easily by noticing places where the chain of reasoning stops unexpectedly.

THE ELECTRONIC CIRCUITS DOMAIN

The circuit purports to be a one-bit full adder, where the first two inputs are the two bits to be added, and the third input is a carry bit. The first output is the sum, and the second output is a carry bit for the next adder.

The goal is to provide an analysis that determines if the circuit is in fact an adder, and that can answer questions about the value of current flow at various points in the circuit.



1. Identify the task

There are many reasoning tasks associated with digital circuits. At the highest level, one analyzes the circuit’s **functionality**.

For example,

Does the circuit actually add properly?

If all the inputs are high, what is the output of gateA2?

Questions about the circuit’s structures are

For example,

What are all the gates connected to the first input terminal?
Does the circuit contain feedback loops?

2. **Assemble the relevant knowledge**

Digital circuits are composed of wires and gates. Signals flow along wires to the input terminals of gates, and each gate produces a signal on the output terminal that flows along another wire.

There are four types of gates: AND, OR, and XOR gates have exactly two input terminals, and NOT gates have one. All gates have exactly one output terminal. Circuits, which are composed of gates, also have input and output terminals.

To reason about functionality and connectivity, we do not need to talk about the wires themselves, or the paths the wires take, or the junctions where two wires come together.

All that matters is the connectivity of terminals—we can say that one output terminal is connected to another input terminal without having to mention the wire that actually connects them.

There are many other factors of the domain that are irrelevant to our analysis, such as the size, shape, color, or cost of the various components.

If our purpose were something other than verifying designs at the gate level, the ontology would be different. For example, if we were interested in debugging faulty circuits, then it would probably be a good idea to include the wires in the ontology, because a faulty wire can corrupt the signal flowing along it.

For resolving timing faults, we would need to include gate delays. If we were interested in designing a product that would be profitable, then the cost of the circuit and its speed relative to other products on the market would be important.

3. **Decide on a vocabulary**

We now know that we want to talk about circuits, terminals, signals, gates, and gate types. The next step is to choose functions, predicates, and constants to name them. We will start from individual gates and move up to circuits.

First, we need to be able to distinguish a gate from other gates. This is handled by naming gates with constants: X1 , *X2*, and so on.

Next, we need to know the type of a gate. A function is appropriate for this: *Type(X1) =XOR.* 'This introduces the constant *XOR* for a particular gate type; the other constants will be called *OR, AND,* and *NOT.*

The *Type* function is not the only way to encode the ontological distinction. We could have used a type predicate: *Type(X1, XOR)* or several predicates, such as *XOR(X1).*

Either of these choices would work fine, but by choosing the function *Type*, we avoid the need for an axiom that says that each individual gate can have only one type. The semantics of functions already guarantees this.

Next we consider terminals. A gate or circuit can have one or more input terminals and one or more output terminals. We could simply name each one with a constant, just as we named gates. Thus, gate X1 could have terminals named *X1In1,X1In2, and X1Out1.*

Names as long and structured as these, however, are as bad as *BearOfVerySmallBrain.* They should be replaced with a notation that makes it clear that *X1Out1* is a terminal for gate X1, and that it is the first output terminal.

A function is appropriate for this; the function *Out(1,X1)* denotes the first (and only) output terminal for gate X1. A similar function *In* is used for input terminals.

The connectivity between gates can be represented by the predicate *Connected,* which takes two terminals as arguments, as in *Connected(Out(l,X1) , In(l,X2)).*

Finally, we need to know if a signal is on or off. One possibility is to use a unary predicate, *On,* which is true when the signal at a terminal is on. This makes it a little difficult, however, to pose questions such as "What are all the possible values of the signals at the following terminals ... ?" We will therefore introduce as objects two "signal values" *1* or *0,* and a function *Signal* which takes a terminal as argument and denotes a signal value.

4. **Encode general rules**

One sign that we have a good ontology is that there are very few general rules that need to be specified. A sign that we have a good vocabulary is that each rule can be stated clearly and concisely. With our example, we need only seven simple rules:

1. If two terminals are connected, then they have the same signal:
   $\forall$ t1,t2 *Connected(t1,t2) ==> Signal(t1)= Signal(t2)*

2. The signal at every terminal is either on or off (but not both):
   $\forall$ *t Signal(t) = 1* V *Signal(t) =0* *1 $\neq$ 0*

3. Connected is a commutative predicate:
   $\forall$ t1,t2 *Connected(t1,t2) <=> Connected(t2,t1)*

4. An OR gate's output is on if and only if any of its inputs are on:
   $\forall$ g *Type(g) = OR => Signal(Out(1,g))=1* $\Leftrightarrow$ $\exists$ n *Signal(In(n , g))=1*

5. An AND gate's output is off if and only if any of its inputs are off:
   $\forall$ g *Type(g)=AND => Signal(Out(l,g))=0<=>* $\exists$ *n Signal(In(n,g)) = 0*

6. An XOR gate's output is on if and only if its inputs are different:
   $\forall$ g *Type(g)=XOR => Signal(Out(1,g)) = 1 <=> Signal(In(l,g)}* $\neq$ *Signal(In(2,g))*

7. A NOT gate's output is different from its input:
   $\forall$ g *(Type(g)=NOT) => Signal(Out(1,g))* $\neq$ *Signal(In(1,g))*

5. **Encode the specific instance**

The circuit is encoded as circuit *C1* with the following description. First, we categorize the gates:

Type(X_1) = *XOR*	*Type*(X_2) = *XOR*
Type(A_l) = *AND*	*Type(A2) = AND*

Type(O1) = OR

Then, the connections between them:

Connected(Out(l,X1),In(l,X2))	*Connected(In(1, C1), In(1,X1))*
Connected(Out(l,X1),In(2,A2))	*Connected(In(l,C1),In(l,A1))*
Connected(Out(1, *A2), In(* 1, O1))	*Connected(In(2,C1),In(2,X1))*
*Connected(Out(*l,*A*1*),In(2, O*1*))*	*Connected(In(2,* C1, *In(2,* A1))
*Connected(Out(*l,*X2), Out(*1, C1)	*Connected(In(3, C1), In(2,X2))*
*Connected(Out(*1, O1), Out(2, C1)	*Connected(In(3, C*1), In(1,A2))

6. **Pose queries to the inference procedure**

What combinations of inputs would cause the first output of C1 (the sum bit) to be 0 and the second output of *C*1 (the carry bit) to be 1?

$\exists$ *i1*,i2, i3 *Signal(In(1, C1))* = i_1 Λ *Signal(In(2,C1))* = i_2 Λ *signal(In(3,* C1) = i_3 Λ *Signal(Out(1,* C1) = 0 Λ *Signal(Out(2,* C1) = 1

The answer are substitutions for the variables i_1, i_2 and i_3 such that the resulting sentence is entailed by the knowledge base. There are three such substitutions:

$\{i_1 / 1, i_2 /1, i_3 /0\}$ $\{ i_1 / 1, i_2 /0, i_3/1\}$ $\{ i_1 / 0, i_2/1, i_3 /1\}$

What are the possible sets of values of all the terminals for the adder circuit?

$\exists$ *i1*,*i*2,i3, o1,o2 *Signal(In(1,C1))* = i_1 *Λ* *Signal(In(2,* C1) = i_2 Λ *Signal(In(3, C1*)) = i_3 Λ *Signal(Out(1,* C1) =o1 Λ *Signal(Out(2,* C1) = o2

This final query will return a complete input/output table for the device, which can be used to check that it does in fact add its inputs correctly. This is a simple example of **circuit verification.**

We can also use the definition of the circuit to build larger digital systems, for which the same kind of verification procedure can be carried out . Many domains are amenable to the same kind of structured knowledge-base development, in which more complex concepts are defined on top of simpler concepts.

7. **Debug the knowledge base**

We can perturb the knowledge base in various ways to see what kinds of erroneous behaviors emerge. For example, suppose we omit the assertion that $1 \neq 0$. Suddenly, the system will be unable to prove any outputs for the circuit, except for the input cases 000 and 110. we can pinpoint the problem by asking for the outputs of each gate. For example, we can ask

$\exists$ *i1*,*i*2,o *Signal(In(1,C1))* = i_1 *Λ* *Signal(In(2,* C1) = i_2 Λ *Signal(Out(*1, X1))

Which reveals that no outputs are known at X1 for the input cases 10 and 01? Then, we look at the axiom for XOR gates, as applied to X1:

$$Signal(Out(1,X1)) = 1 \iff Signal(In(1, X1)) \neq Signal(In(2, X1))$$

If the inputs are known to be, say, 1 and 0, then this reduces to

$$Signal(Out(1,X1)) = 1 \iff 1 \neq 0$$

Now the problem is apparent: the system is unable to infer that $Signal(Out(1,X1)) = 1$, so we need to tell it that $1 \neq 0$

11. KNOWLEDGE REPRESENTATION

Internal Representation

central for „reasoning“: internal representation and symbol manipulation.

- Representation in general: An idealized world description (not necesserily symbolic)
- Internal symbolic representation: requires a common symbol language, in which an agent can express and manipulate propositions about the world.
- good choicefor symbolic representations are languages of logic, however, some preparations have to be made...

Symbolic representations must be unique regarding several aspects:

• referential • semantic • functional

- **Make References Explicit**

Natural language often is ambiguous:

The chair was placed on the table. It was broken.

Is represented as

The chair (r1) was placed on the table (r2).It (r1) was broken.

(Now it becomes obvious what was broken.)

- **Refrential Uniqueness**
  - Symbolic representations must explicitly define relations for entity references!
  - i.e., all ambiguity with respect to entities must be eliminated in the internal representation:
  - all individuals get a unique name
  - this means only one individual per name

Hence: instead of multiple "Daves": dave-1, dave-2 etc.

Such unique names are denoted as ***instances* or *tokens*.**

- **Functional Uniqueness**

  Jack caught a ball. **[catch-object]**

  Jack caught a cold. **[catch-illness]**

  Different symbols imply different semantics

  (even if their linguistic roots might be the same):

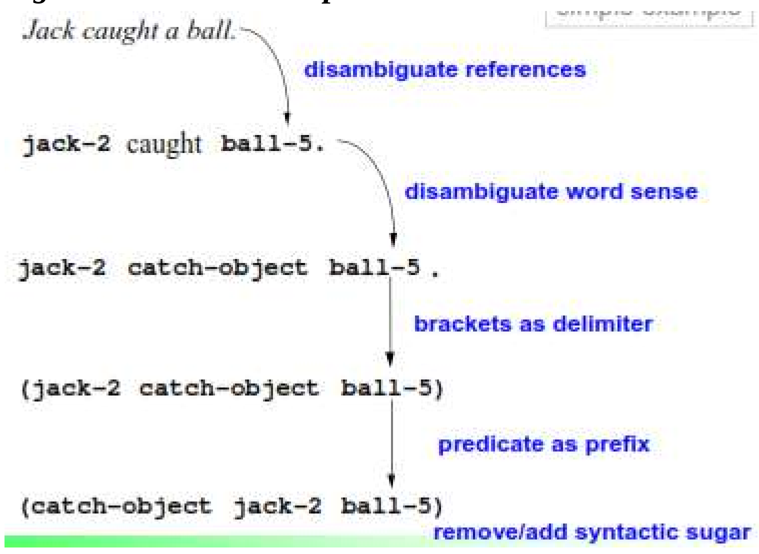
  For example, who caught a cold must sneeze

- **Internal representations must uniquely express the functional roles!**

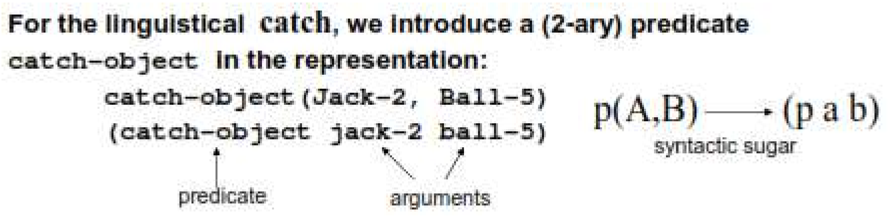
Petra catches the ball. The ball Petra catches.
The ball is caught by Petra.

Who is the catcher? Who or what is the caught object?

From Linguistic Sentence to Representation



Predicates, Logic Sentences, Assertions

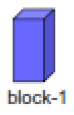

For the linguistical catch, we introduce a (2-ary) predicate catch-object in the representation:

```
catch-object(Jack-2, Ball-5)
(catch-object jack-2 ball-5)
```

- A logic sentence defines a fact about one or multiple entities, in this case a catch relation between one Jack and a specific ball.
- Assertions are logic sentences which we take as given facts (as elements of an actual internal representation)

Linguistic Sentence and Representation

In general, a linguistic sentence is represented by multiple logic sentences:



Jack caught a blue block.

(catch-object jack-1 block-1)
(inst block-1 block)
(color block-1 blue)

- Processes operating on internal representations are used to deduct derive new facts from known facts: *Inference*
- Commonly used inference concept and term: *Deduction*
- Such processes can be modeled in higher order logic. (Normally we use first order logic.)

Slot-Assertion-Notation

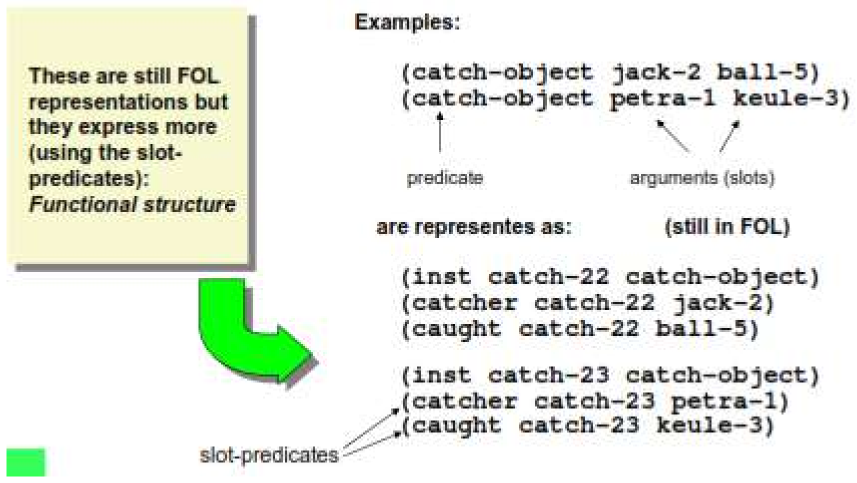
A slot is an attribute value pair in its simplest form

A filler is a value that a slot can take

A weak slot and filler structure does not consider the content of the representation.

- It enables attribute values to be retrieved quickly
    - assertions are indexed by the entities
    - binary predicates are indexed by first argument. *E.g. team(Mike-Hall , Cardiff).*
- Properties of relations are easy to describe .
- It allows ease of consideration as it embraces aspects of object oriented programming.
- A *slot* is an attribute value pair in its simplest form.
- A *filler* is a value that a slot can take -- could be a numeric, string (or any data type) value or a pointer to another slot

Reason: To express functional relations


These are still FOL representations but they express more (using the slot-predicates): *Functional structure*

Examples:

```
(catch-object jack-2 ball-5)
(catch-object petra-1 keule-3)
```

are represented as: (still in FOL)

```
(inst catch-22 catch-object)
(catcher catch-22 jack-2)
(caught catch-22 ball-5)

(inst catch-23 catch-object)
(catcher catch-23 petra-1)
(caught catch-23 keule-3)
```

<u>**Slot-and-Filler-Notation (–>Frames)**</u>

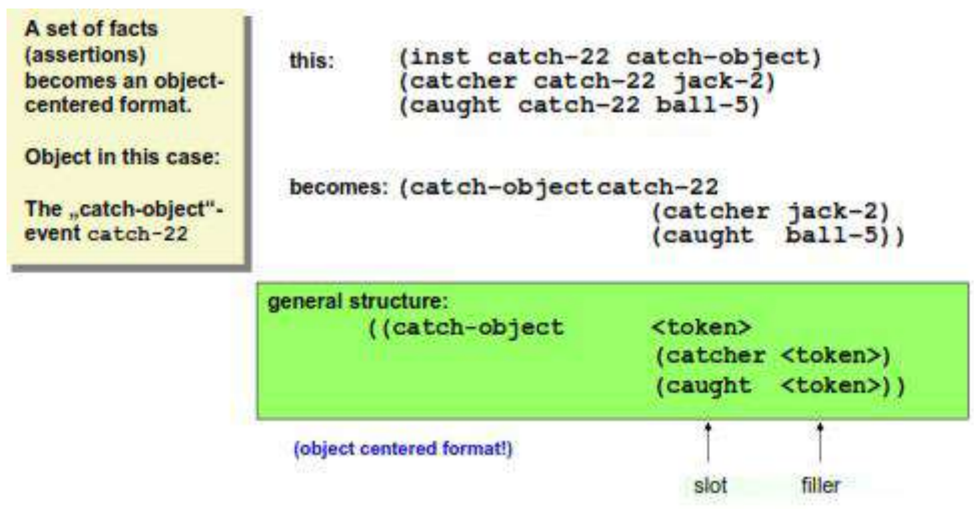
Different slot-assertions are combined to provide a structured expression

A *frame* is a collection of attributes or slots and associated values that describe some real world entity. Frames on their own are not particularly helpful but frame systems are a powerful way of encoding information to support reasoning.

1Frames as Sets and Instances

Set theory provides a good basis for understanding frame systems. Each frame represents:

- a class (set), or
- an instance (an element of a class).



A set of facts (assertions) becomes an object-centered format.

Object in this case:

The „catch-object"-event `catch-22`

```
this:    (inst catch-22 catch-object)
         (catcher catch-22 jack-2)
         (caught catch-22 ball-5)

becomes: (catch-objectcatch-22
                             (catcher jack-2)
                             (caught  ball-5))
```

```
general structure:
       ((catch-object      <token>
                           (catcher <token>)
                           (caught  <token>))
```

(object centered format!)

slot

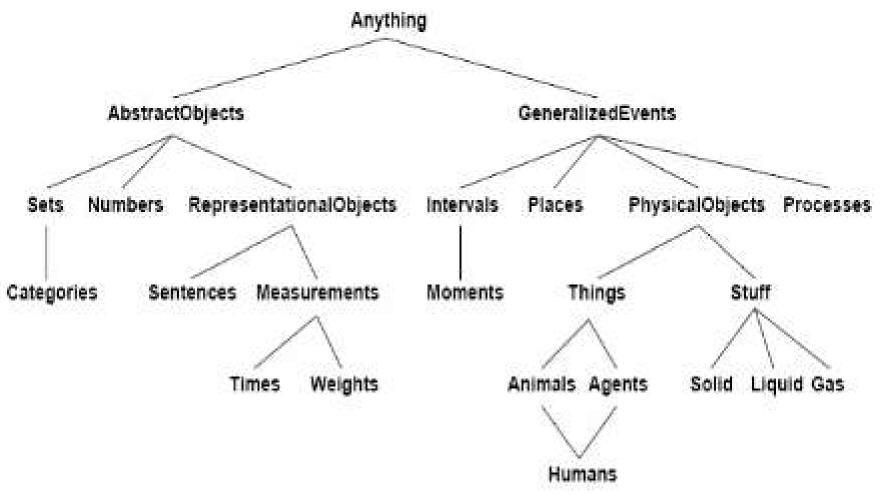
filler

12. <u>ONTOLOGICAL ENGINEERING</u>

Representing these abstract concepts is called **ontological engineering**.

- How to create more general and flexible representations.
- Concepts like actions, time, physical object and beliefs
- Operates on a bigger scale than K.E.
- Define general framework of concepts
- Upper ontology
- Limitations of logic representation
- Red, green and yellow tomatoes: exceptions and uncertainty

The upper ontology of the world



For example, we will define what it means to be a physical object and the detail of different types of objects—robots, televisions, books, or whatever—can be filled in later.
The general framework of concepts is called an **upper ontology,** because of the convention of drawing graphs with the general concepts at the top and the more specific concepts below them

For example, although "tomatoes are red" is a useful rule, some tomatoes are green, yellow, or orange. Similar exceptions can be found to almost all the general statements in this chapter.

The ability to handle exceptions and uncertainty is extremely important, but is orthogonal to the task of understanding the general ontology.

For example, time is omitted completely. Signals are fixed, and do not propagate. The structure of the circuit remains constant. If we wanted to make this more general, consider signals at particular times, and include the wire lengths and propagation delays. This would allow us to simulate the timing properties of the circuit, and indeed such simulations are often carried out by circuit designers.

We could also introduce more interesting classes of gates, for example by describing the technology (TTL, MOS, CMOS, and so on) as well as the input/output specification. If we wanted to discuss reliability or diagnosis, we would include the possibility that the structure of the circuit or the properties of the gates might change spontaneously. To account for stray capacitances, we would need to move from a purely topological representation of connectivity to a more realistic description of geometric properties.

Difference with special-purpose ontologies

- A general-purpose ontology should be applicable in more or less any special-purpose domain.
    - Add domain-specific axioms

A more general ontology, better suited for the real world, would allow for simultaneous changes extended over time. We also used a Pit predicate to say which squares have pits. We could have allowed for different kinds of pits by having several individuals belonging to the class of pits, each having different properties.

For any special-purpose ontology, it is possible to make changes like these to move toward greater generality. There are two major characteristics of general- purpose ontologies that distinguish them from collections of special-purpose ontologies:

• A general-purpose ontology should be applicable in more or less any special-purpose domain. This means that, as far as possible, no representational issue can be finessed or brushed under the carpet.

• In any sufficiently demanding domain, different areas of knowledge must be unified, because reasoning and problem solving could involve several areas simultaneously.

- What do we need to express?

Categories, Measures, Composite objects, Time, Space, Change, Events, Processes, Physical Objects, Substances, Mental Objects, Beliefs

13. **Categories and Objects**

- KR requires the organisation of objects into **categories**
- Interaction at the level of the object
- Reasoning at the level of categories

The **organization of objects into categories** is a vital part of knowledge representation. Although interaction with the world takes place at the level of individual objects, much reasoning takes place at the level of categories.

For example, a shopper might have the goal of buying a basketball, rather than a particular basketball such as BB_9. Categories also serve to make predictions about objects once they are classified.

One infers the presence of certain objects from perceptual input, infers category membership from the perceived properties of the objects, and then uses category information to make predictions about the objects.

For example, from its green, mottled skin, large size, and avoid shape, one can infer that an object is a watermelon; from this, one infers that it would be useful for fruit salad.

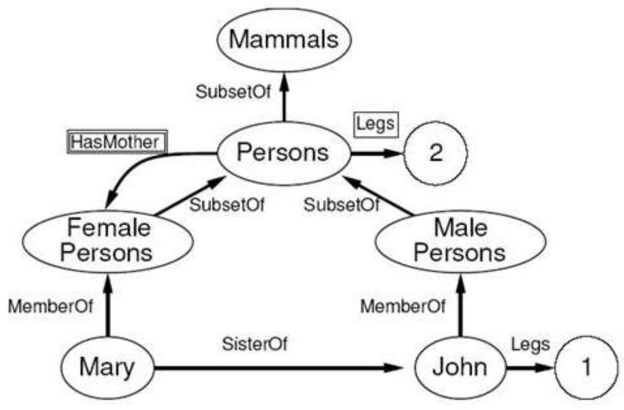
- Categories play a role in predictions about objects
- Based on perceived properties
- Categories can be represented in two ways by FOL
- Predicates: apple(x)
- *Reification* of categories into objects: apples

Categories serve to organize and simplify the knowledge base through inheritance. If we say that all instances of the category Food are edible, and if we assert that Fruit is a subclass of Food and Apples is a subclass of Fruit, then we know that every apple is edible.

- Category = set of its members

Category organization

- Relation = *inheritance*:
- All instance of food are edible, fruit is a subclass of food and apples is a subclass of fruit then an apple is edible.
- Defines a taxonomy



We say that the individual apples inherit the property of edibility, in this case from their membership in the Food category.

Subclass relations organize categories into a taxonomy, or taxonomic hierarchy. Taxonomies have been used explicitly for centuries in technical fields.

For example, systematic biology aims to provide a taxonomy of all living and extinct species; tax authorities and other government departments have developed extensive taxonomies of occupations and commercial products. Taxonomies are also an important aspect of general commonsense knowledge.

First-order logic makes it easy to state facts about categories, either by relating objects to categories or by quantifying over their members:

FOL and categories

- An **object** is a member of a category
- MemberOf(BB_{12},Basketballs)
- A **category** is a subclass of another category
- SubsetOf(Basketballs,Balls)
- All members of a category have some properties
- $\forall$ x (MemberOf(x,Basketballs) $\Rightarrow$ Round(x))
- All members of a category can be recognized by some properties
- $\forall$ x (Orange(x) $\wedge$ Round(x) $\wedge$ Diameter(x)=9.5in $\wedge$ MemberOf(x,Balls)

$\Rightarrow$ MemberOf(x,BasketBalls))

- A category as a whole has some properties
- MemberOf(Dogs,DomesticatedSpecies)

Relations between categories

- Two or more categories are ***disjoint*** if they have no members in common:
- Disjoint(s)$\Leftrightarrow$($\forall$ c_1,c_2 $c_1 \in s \wedge c_2 \in s \wedge c_1 \neq c_2 \Rightarrow$ Intersection(c_1,c_2) ={})
- Example: Disjoint({animals, vegetables})

- A set of categories *s* constitutes an ***exhaustive decomposition*** of a category *c* if all members of the set *c* are covered by categories in *s*:
- E.D.(s,c) $\Leftrightarrow$ ($\forall$ i i $\in$ c $\Rightarrow \exists$ c_2 $c_2 \in s \wedge i \in c_2$)
- Example:

ExhaustiveDecomposition({Americans, Canadian, Mexicans}, NorthAmericans)

Relations between categories

- A ***partition*** is a disjoint exhaustive decomposition:
- Partition(s,c) $\Leftrightarrow$ Disjoint(s) $\wedge$ E.D.(s,c)
- Example: Partition({Males,Females},Persons).
- Is ({Americans,Canadian, Mexicans},NorthAmericans)

a partition?

- No! There might be dual citizenships.
- Categories can be defined by providing necessary and sufficient conditions for membership
- $\forall$ x Bachelor(x) $\Leftrightarrow$ Male(x) $\wedge$ Adult(x) $\wedge$ Unmarried(x)

Natural kinds

- Many categories have no clear-cut definitions

(e.g., chair, bush, book).

- Tomatoes: sometimes green, red, yellow, black. Mostly round.
- One solution: subclass using category *Typical(Tomatoes).*
- Typical(c) $\subseteq$ c

- ∀ x, x ∈ Typical(Tomatoes) ⇒ Red(x) ∧ Spherical(x).
- We can write down useful facts about categories without providing exact definitions.

- Wittgenstein (1953) gives an exhaustive summary about the problems involved when exact definitions for natural kinds are required in his book "Philosophische Untersuchungen".
- What about "bachelor"? Quine (1953) challenged the utility of the notion of *strict definition*. We might question a statement such as "the Pope is a bachelor".

Physical composition

The idea that one object can be part of another is a familiar one. One's nose is part of one's head, Romania is part of Europe. We use the general part of relation to say that one thing is part of another. Objects can be grouped into PartOf hierarchies, reminiscent of the Subset hierarchy:

- One object may be part of another:
- PartOf(Bucharest,Romania)
- PartOf(Romania,EasternEurope)
- PartOf(EasternEurope,Europe)
- The PartOf predicate is transitive (and reflexive), so we can infer that

PartOf(Bucharest,Europe)

- More generally:
- ∀ x PartOf(x,x)
- ∀ x,y,z PartOf(x,y) ∧ PartOf(y,z) ⇒ PartOf(x,z)
- Often characterized by structural relations among parts.
- E.g. Biped(a) ⇒

$$\begin{gathered}(\exists l_1, l_2, b)(Leg(l_1) \wedge Leg(l_2) \wedge Body(b) \wedge \\ PartOf(l_1, a) \wedge PartOf(l_2, a) \wedge PartOf(b, a) \wedge \\ Attached(l_1, b) \wedge Attached(l_2, b) \wedge \\ l_1 \neq l_2 \wedge (\forall l_3)(Leg(l_3) \Rightarrow (l_3 = l_1 \vee l_3 = l_2)))\end{gathered}$$

14. Measurements

In both scientific and common sense theories of the world, objects have height, mass, cost, and so on. The values that we assign for these properties are called **measures**

- Objects have height, mass, cost, ...

Values that we assign to these are **measures**

- Combine Unit functions with a number:

Length(L_1) = Inches(1.5) = Centimeters(3.81).

- Conversion between units:

∀ i Centimeters(2.54 x i)=Inches(i).

- Some measures have no scale: Beauty, Difficulty, etc.
- Most important aspect of measures:

they are **orderable**.

- Don't care about the actual numbers.

(An apple can have deliciousness .9 or .1.)

There are some properties that are **intrinsic**: they belong to the very substance of the object, rather than to the object as a whole.

extrinsic properties are the opposite: properties such as weight, length, shape, function, and so on are not retained under subdivision.

A class of objects that includes in its definition only intrinsic properties is then a substance, or mass noun; a class that includes any extrinsic properties in its definition is a count noun.

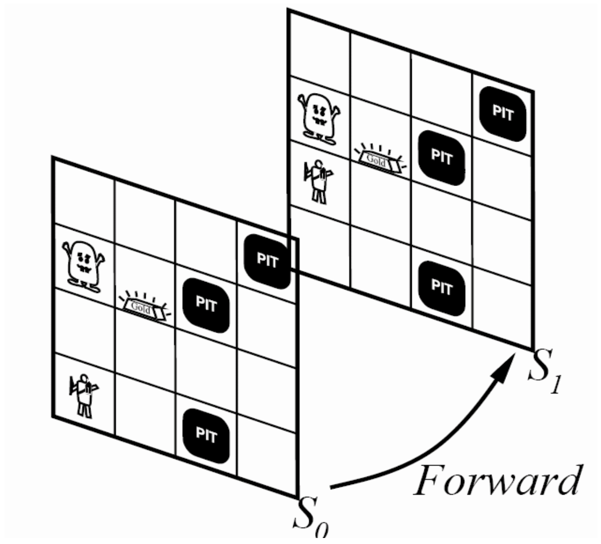
The category **Stuff** is the most general substance category, specifying no intrinsic properties.
The category **Thing** is the most general discrete object category, specifying no extrinsic properties.

All physical objects belong to both categories, so the categories are coextensive—they refer to the same entities.

Actions, events and situations

Reasoning about outcome of actions is central to KB-agent.

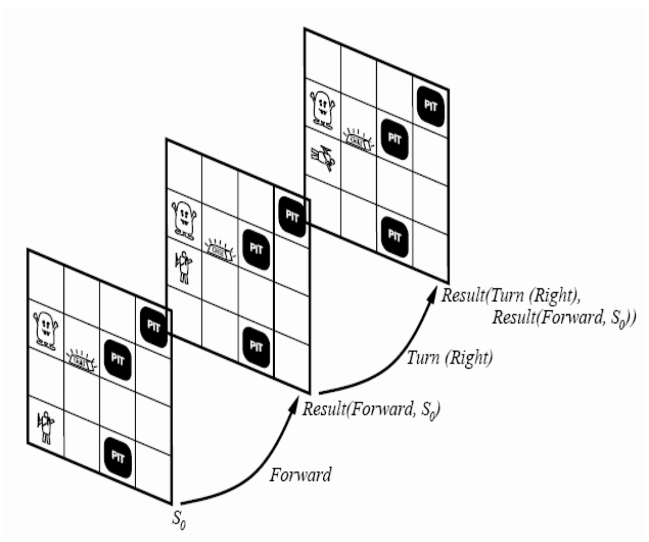
- How can we keep track of location in FOL?
- Remember the multiple copies in PL.
- Representing time by situations (states resulting from the execution of actions).
- Situation calculus



<u>•Situation calculus:</u>

- Actions are logical terms
- Situations are logical terms consiting of
- The initial situation I

- All situations resulting from the action on I (=*Result(a,s)*)
- **Fluents** are functions and predicates that vary from one situation to the next.
- E.g. $\neg Holding(G_1, S_0)$
- **Eternal predicates** are also allowed
- E.g. $Gold(G_1)$



- Results of action sequences are determined by the individual actions.
- *Projection task:* an SC agent should be able to deduce the outcome of a sequence of actions.
- *Planning task:* find a sequence that achieves a desirable effect

Describing change

Each action is described by two axioms:

1. Possibility axiom that says when it is possible to execute the action, and
2. Effect axiom that says effect axiom what happens when a possible action is executed.

   We will use Poss(a, s) to mean that it is possible to execute action a in situation s.

- Simples Situation calculus requires two axioms to describe change:
- Possibility axiom: when is it possible to do the action

$At(Agent,x,s) \land Adjacent(x,y) \Rightarrow Poss(Go(x,y),s)$

- Effect axiom: describe changes due to action

$Poss(Go(x,y),s) \Rightarrow At(Agent,y,Result(Go(x,y),s))$

Frame problem

- What stays the same?
- Frame problem: how to represent all things that stay the same?
- Frame axiom: describe non- changes due to actions

$At(o,x,s) \land (o \neq Agent) \land \neg Holding(o,s) \Rightarrow At(o,x,Result(Go(y,z),s))$

Representational frame problem

- If there are F fluents and A actions then we need AF frame axioms to describe other objects are stationary unless they are held.
- We write down the effect of each actions

- Solution; describe how each fluent changes over time
- Successor-state axiom:

$$Pos(a,s) \Rightarrow (At(Agent,y,Result(a,s)) \Leftrightarrow (a = Go(x,y)) \vee$$
$$(At(Agent,y,s) \wedge a \neq Go(y,z))$$

- Note that next state is completely specified by current state.
- Each action effect is mentioned only once

To solve the inferential frame problem, we have two possibilities.

1. First, we could discard situation calculus and invent a new formalism for writing axioms. This has been done with formalisms such as the fluent calculus.
2. Second, we could alter the inference mechanism to handle frame axioms more efficiently.

We need to say that an implicit effect of an agent moving from x to y is that any gold it is carrying will move too (as will any ants on the gold, any bacteria on the ants, etc.). Dealing with implicit effects is called the **ramification problem**.

Other problems

- How to deal with secondary (implicit) effects?
- If the agent is carrying the gold and the agent moves then the gold moves too.
- Ramification problem
- How to decide EFFICIENTLY whether fluents hold in the future?
- Inferential frame problem.
- Extensions:
- Event calculus (when actions have a duration)
- Process categories

15. MENTAL EVENTS AND OBJECTS

- So far, KB agents can have beliefs and deduce new beliefs
- What about knowledge about beliefs? What about knowledge about the inference proces?
- Requires a model of the mental objects in someone’s head and the processes that manipulate these objects.
- Relationships between agents and mental objects: believes, knows, wants, …
- Believes(Lois,Flies(Superman)) with Flies(Superman) being a function … a candidate for a mental object (reification).
- Agent can now reason about the beliefs of agents

The agents we have constructed so far have beliefs and can deduce new beliefs. Yet none of them has any knowledge about beliefs or about deduction. For single-agent domains, knowledge about one's own knowledge and reasoning processes is useful for controlling inference.

For example, if one knows that one does not know anything about Romanian geography, then one need not expend enormous computational effort trying to calculate the shortest path from Arad to Bucharest. One can also reason about one's own knowledge in order to construct plans that will change it—for example by buying a map of Romania. In multiagent domains, it becomes important for an agent to reason about the mental states of the other agents.

For example, a Romanian police officer might well know the best way to get to Bucharest, so the agent might ask for help.

In essence, what we need is a model of the mental objects that are in someone's head (or something's knowledge base) and of the mental processes that manipulate those mental objects. The model should be faithful, but it does not have to be detailed.

We will be happy to conclude that the Romanian police officer will tell us how to get to Bucharest if he or she knows the way and believes we are lost.

A formal theory of beliefs

We begin with the relationships between agents and "mental objects"—relationships such as Believes, Knows, and Wants.

Relations of this kind are called **propositional attitudes**, because they describe an attitude that an agent can take toward a proposition. Suppose that Lois believes something—that is, Believes(Lois,x). What kind of thing is xl Clearly, x cannot be a logical sentence.

If Flies (Superman) is a logical sentence, we can't say Believes(Lois, Flies [Superman)), because only terms (not sentences) can be arguments of predicates. But if Flies is a function, then Flies [Superman) is a candidate for being a mental object, and Believes can be a relation between an agent and a propositional fluent. Turning a proposition into an object is called **reification.**

This appears to give us what we want: the ability for an agent to reason about the beliefs of agents. Unfortunately, there is a problem with that approach: If Clark and Superman are one and the same (i.e., Clark = Superman) then Clark's flying and Superman's flying are one and the same event category, i.e., Flies(Clark) = Flies (Superman). Hence, we must conclude that if Lois believes that Superman can fly, she also believes that Clark can fly, even if she doesn't believe that Clark is Superman. That is,

$$(\text{Superman} = \text{Clark}) \models (\text{Believes(Lois, Flies (Superman))} \Leftrightarrow \text{Believes [Lois, Flies (Clark)))}\ .$$

There is a sense in which this is right: Lois does believe of a certain person, who happens to be called Clark sometimes, that that person can fly. But there is another sense in which it is wrong: if you asked Lois "Can Clark fly?" she would certainly say no. Reified objects and events work fine for the first sense of Believes, but for the second sense we need to reify descriptions of those objects and events, so that Clark and Superman can be different descriptions (even though they refer to the same object).

Technically, the property of being able to substitute a term freely for an equal term is called referential transparency. In first-order logic, every relation is referentially transparent. We would like to define Believes (and the other prepositional attitudes) as relations whose second argument is referentially opaque—that is, one cannot substitute an equal term for the second argument without changing the meaning.

There are two ways to achieve this.

1. The first is to use a different form of logic called **modal logic**, in which prepositional attitudes such as Believes and Knows become **modal operators** that are referentially opaque.

2. The second approach, which we will pursue, is to achieve effective opacity within a referentially transparent language using a **syntactic theory** of mental objects. This means that mental objects are represented by **strings**. The result is a crude model of an agent's knowledge base as consisting of strings that represent sentences believed by the agent. A string is just a complex term denoting a list of symbols, so the event Flies(Clark) can be represented by the list of characters [F, l, i, e, s, (, C, l, a, r, k,)], which we will abbreviate as a quoted string, “Flies(Clark)". The syntactic theory includes a **unique string axiom** stating that strings are identical if and only if they consist of identical characters. In this way, even if Clark = Superman, we still have "Clark" ≠ "Superman".

We start by defining Den as the function that maps a string to the object that it denotes and Name as a function that maps an object to a string that is the name of a constant that denotes the object.

For example, the denotation of both "Clark" and "Superman" is the object referred to by the constant symbol ManOfSteel, and the name of that object within the knowledge base could be either "Superman", "Clark", or some other constant, such as "In":

Den("Clark") = ManOfSteel Λ Den("Superman") = ManOfSteel .
Name{ManOfSteel) = "X_{11}" .

The next step is to define inference rules for logical agents. For example, we might want to say that a logical agent can do Modus Ponens: if it believes p and believes p => q, then it will also believe q. The first attempt at writing this axiom is

LogicalAgent(a) Λ Believes(a,p) Λ Believes(a,"p=> q") => Believes(a,q) .

But this is not right because the string "p => q" contains the letters 'p' and 'q' but has nothing to do with the strings that are the values of the variables p and q. The correct formulation is

LogicalAgent(a) Λ Believes(a,p) Λ Believes(a, Concat(p,"=>", q)) => Believes(a,q) .

Where Concat is a function on strings that concatenates their elements. We will abbreviate Concat(p,"=>",q) as "p=>q". That is, an occurrence of $\underline{x}$ within a string is **unquoted**, meaning that we are to substitute in the value of the variable x.

Lisp programmers will recognize this as the comma/backquote operator, and Perl programmers will recognize it as $-variable interpolation.

Once we add in the other inference rules besides Modus Ponens, we will be able to answer questions of the form "given that a logical agent knows these premises, can it draw that conclusion?" Besides the normal inference rules, we need some rules that are specific to belief.

For example, the following rule says that if a logical agent believes something, then it
believes that it believes it.

LogicalAgent(a) Λ Believes(a,p) => Believes(a,"Believes$\underline{\text{(Name(a), p}}$)") .

Now, according to our axioms, an agent can deduce any consequence of its beliefs infallibly. This is called **logical omniscience**. A variety of attempts have been made to define limited rational agents, which can make a limited number of deductions in a limited time.

None is completely satisfactory, but these formulations do allow a highly restricted range of predictions about limited agents.

THE INTERNET SHOPPING WORLD

- A Knowledge Engineering example
- An agent that helps a buyer to find product offers on the internet.
- IN = product description (precise or ¬precise)
- OUT = list of webpages that offer the product for sale.
- Environment = WWW
- Percepts = web pages (character strings)
- Extracting useful information required

- Find relevant product offers

RelevantOffer(page,url,query) ⇔ *Relevant(page, url, query)* ∧ *Offer(page)*

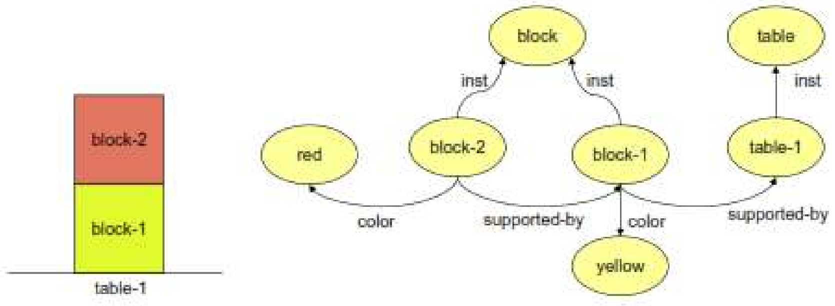
- Write axioms to define Offer(x)
- Find relevant pages: Relevant(x,y,z) ?
- Start from an initial set of stores.
- What is a relevant category?
- What are relevant connected pages?
- Require rich category vocabulary.
- Synonymy and ambiguity
- How to retrieve pages: *GetPage(url)*?
- Procedural attachment
- Compare offers (information extraction)

16. REASONING SYSTEMS FOR CATEGORIES

- How to organize and reason with categories?
- Semantic networks
- Visualize knowledge

- Efficient algorithms for category membership inference
- Description logics
- Formal language for constructing and combining category definitions
- Efficient algorithms to decide subset and superset relationships between categories

REPRESENTATION OF A SCE

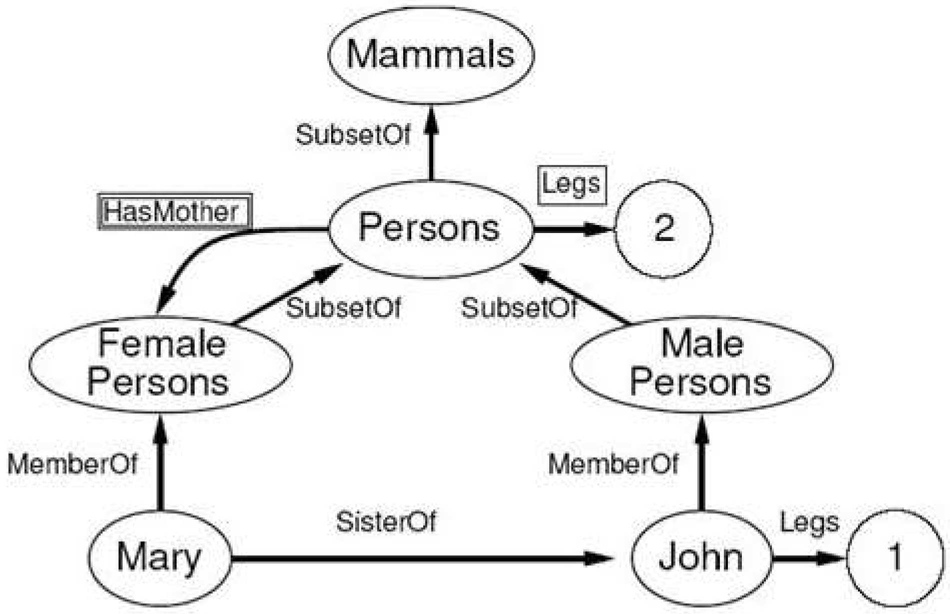


(inst block-2 block)
(color block-2 red)
(supported-by block-2 block-1) (

inst block-1 block)
(color block-1 yellow)
(supported-by block-1 table-1)
as set of logic expressions
• as Semantic Net

16. SEMANTIC NETWORKS
A Semantic Net : is a formal graphic language representing facts about entities in some world about which we wish to reason. The meaning of a concept comes from the ways in which it is connected to other concepts. In semantic net, information is represented as a set of nodes connected to each other by set of labeled arcs, which represent relationships among the nodes.

- Logic vs. semantic networks
- Many variations
- All represent individual objects, categories of objects and relationships among objects.
- Allows for inheritance reasoning
- Female persons inherit all properties from person.
    - Cfr. OO programming.
- Inference of inverse links
- SisterOf vs. HasSister



<u>**Alternative Notations**</u>
Semantic Nets (a.k.a. „associative nets) and FOL sentences represent same information in different formats:

Nodes correspond to terms marked out directed edges correspond to predicates

They are alternative notations for the same content, not in principle different representations!

Missing existential quantifier Functions (extensions exist) Semantic nets additionally provide pointers (and sometimes back pointers) which allow easy and high-performance information access (e.g., to instances): INDEXING

ISA-Hierarchy and Inheritance

- Key concept in the tradition of semantic nets
- Instances inherit properties which we attribute to sets of individuals (classes).
- This can be propagates along the complete isa hierarchy
- Inheritance of properties

Reason: Knowledge representation economy

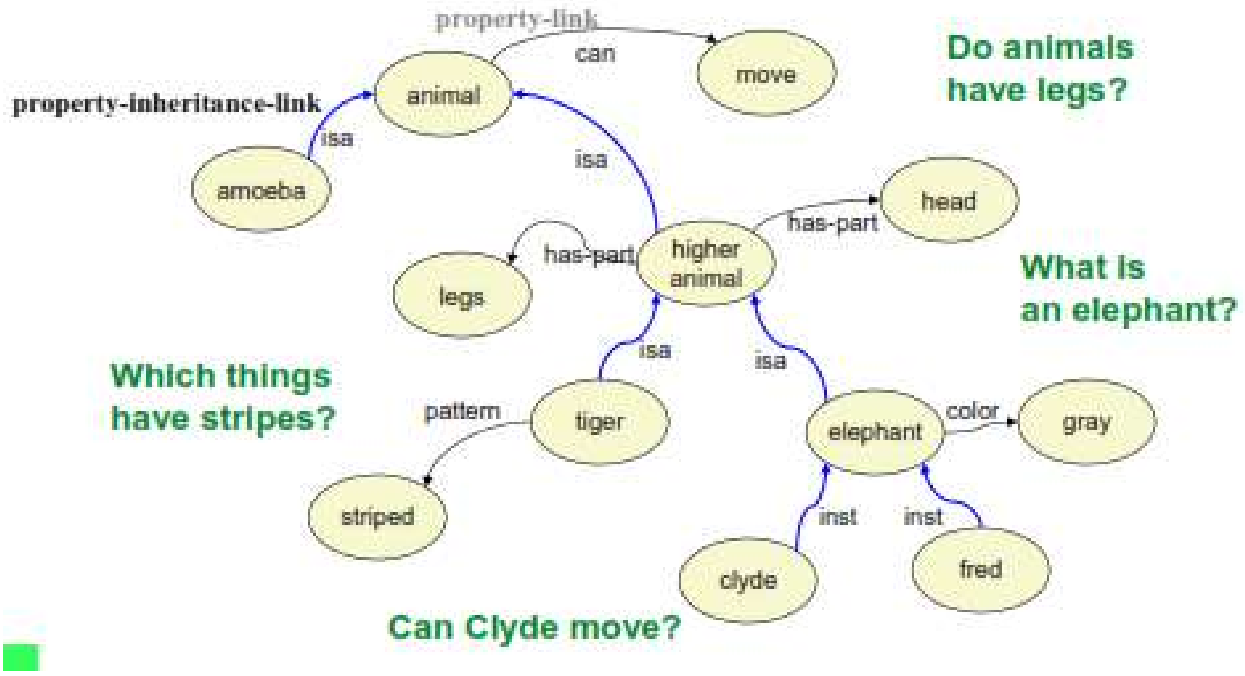
- Search along isa- and inst-links to access information not

"instance of" directly associated (using inheritance)

„Instanz von"

• inst:	$\in$	**member of**
• isa:	$\subseteq$	**subset of**

<u>Example of an ISA-Hierarchy</u>

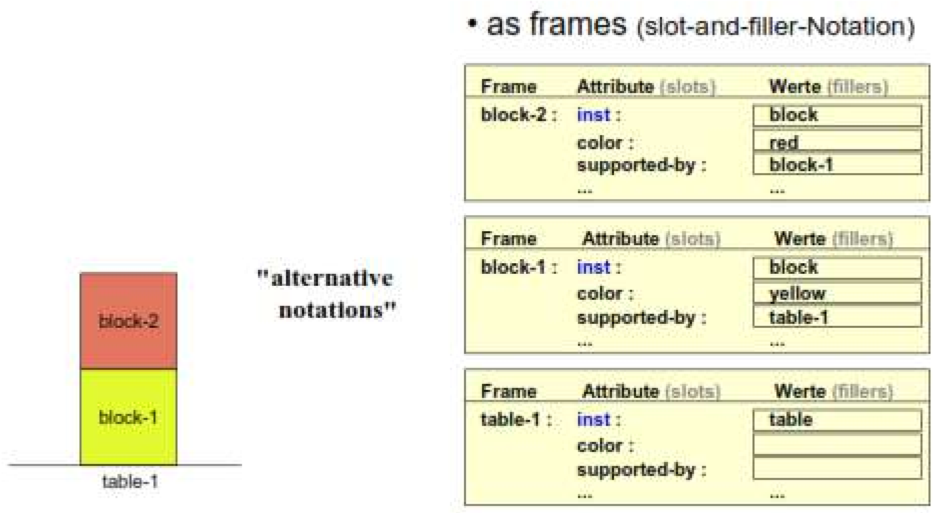


🟩

Drawbacks

- Links can only assert binary relations
- Can be resolved by reification of the proposition as an event
- Representation of default values
- Enforced by the inheritance mechanism.

Representation of a Scene
(inst block-2 block) (color block-2 red)
(supported-by block-2 block-1) (inst block-1 block)
(color block-1 yellow) (supported-by block-1 table-1) (inst table-1 table)



• as frames (slot-and-filler-Notation)

Frame	Attribute (slots)	Werte (fillers)
block-2 :	inst :	block
	color :	red
	supported-by :	block-1
	...	...

Frame	Attribute (slots)	Werte (fillers)
block-1 :	inst :	block
	color :	yellow
	supported-by :	table-1
	...	...

Frame	Attribute (slots)	Werte (fillers)
table-1 :	inst :	table
	color :	
	supported-by :	
	...	...

PRATHYUSHA ENGINEERING COLLEGE
Department of Computer Science and Engineering
VI Semester (EVEN 2019-20)
CS8691ARTIFICIAL INTELLIGENCE

UNIT IV SOFTWARE AGENTS 9
Architecture for Intelligent Agents –Agent communication –Negotiation and Bargaining –Argumentation among Agents –Trust and Reputation in Multi-agent systems.

INTRODUCTION

Intelligent Agent is an autonomous entity that exists in an environment and acts in a rational way.

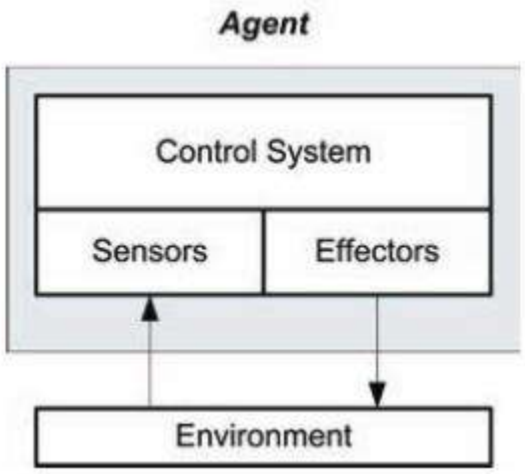


FIGURE 11.1: **The fundamental anatomy of an agent.**

An agent is commonly made up of a number of elements. These include one or more sensors that are used to perceive the environment, one or more effectors that manipulate the environment, and a control system. The control system provides a mapping from sensors to effectors, and provides intelligent (or rational) behaviour.

This can also be applied to other types of agents, both virtual and physical.
EXAMPLE: web spider

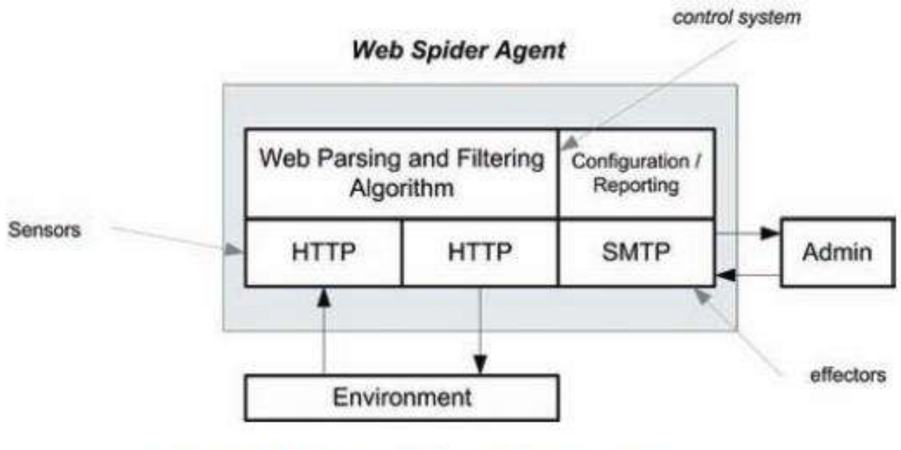


FIGURE 11.2: **Web spider as an agent.**

Virtual agent that gathers and filters information for another party.
A web spider uses a primary sensor of the HyperText Transport Protocol, or HTTP, as a means to gather data from web pages.

Its control system is an application, which can be written in almost any language, that drives the behavior of the web spider.
This behavior includes web-data parsing and filtering. The web spider can identify new links to follow to collect additional information, and use the HTTP protocol to navigate the web environment.
Finally, the web spider can communicate with a managing user through email using the Simple Mail Transport Protocol, or SMTP.
The user can configure the web spider for collection, navigation, or filtering, and also receive emails indicating its current status
The web spider is an intermediary agent for web-data gathering and filtering for a user. The web spider acts on the user's behalf for data gathering, given a set of constraints from the user

ROBOT
A robot can also be considered an agent.
A robot includes a variety of sensors including vision (cameras, ultrasonic transducers, infrared detectors), hearing (microphones), touch (bump sensors), as well as other types of sensors for pressure, temperature, and movement detection (accelerometers).
Effectors include motors (to drive wheels, tracks, or limbs), and a speaker for sound vocalization. A robot can also include a number of other effectors that can manipulate the environment, such as a vacuum, a water pump, or even a weapon.

TAXONOMY OF AGENTS

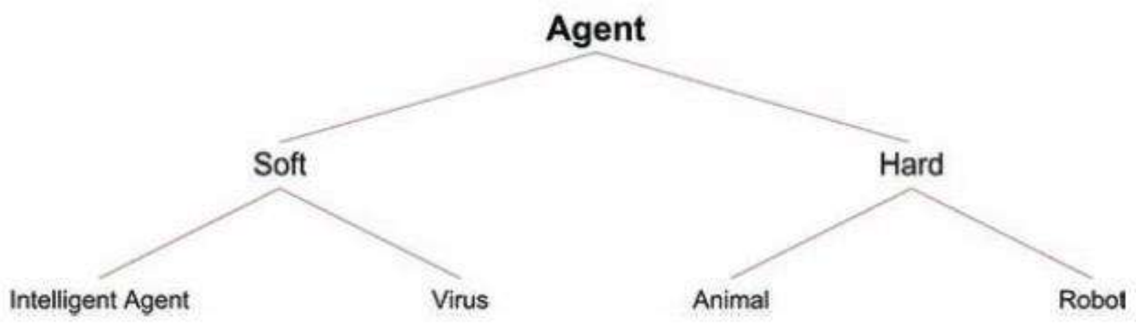
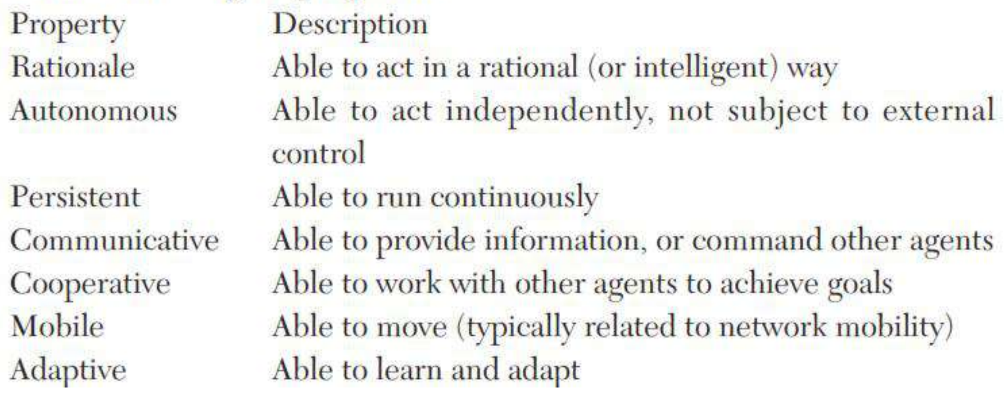


FIGURE 11.3: **Simple agent taxonomy.**

<u>AGENT PROPERTIES</u>
an *agent* is that it utilizes one or more properties that exhibit some type of intelligence

Table 11.1: Agent properties.



Property	Description
Rationale	Able to act in a rational (or intelligent) way
Autonomous	Able to act independently, not subject to external control
Persistent	Able to run continuously
Communicative	Able to provide information, or command other agents
Cooperative	Able to work with other agents to achieve goals
Mobile	Able to move (typically related to network mobility)
Adaptive	Able to learn and adapt

Rationale
The property of rationality simply means that the **agent does the right thing at the right time**, given a known outcome. This depends on the actions that are available to the agent (can it achieve the best outcome), and also how the agent's performance is measured.

Autonomous

Autonomy simply means that the agent is able to navigate its environment without guidance from an external entity (such as a human operator). The autonomous agent can therefore seek out goals in its environment, whether to sustain itself or solve problems. An example of an autonomous agent is the remote agent that rode along in NASA's Deep Space 1 spacecraft.

Persistent

Persistence implies that the agent exists over time and continuously exists in its environment. This property can also imply that the agent is stateful in conditions where the agent must be serialized and moved to a new location (as would be the case for mobile agents).

Communicative

An agent having the ability to communicate provides obvious advantages to agent systems. Agents can communicate with other agents to provide them with information, or communicate with users (for whom the agent represents). An example of agent communication was shown in Figure 11.2.

AGENT ENVIRONMENTS

Whether an agent's environment is the Internet, virtual landscape of a game, or the unique space of a problem environment, all environments share a common set of characteristics

Property	**Description**
Observability	Are all elements visible to the agent?
Dynamic	Does the environment change (dynamic) or does it stay the same (static) and change only when the agent performs an action to initiate change?
Deterministic	Does the state of the environment change as the agent expects after an action (deterministic), or is there some randomness to the environment change from agent actions (stochastic)?
Episodic	Does the agent need prior understanding of the environment (prior experience) to take an action (sequential), or can the agent perceive the environment and take an action (episodic)?
Continuous	Does the environment consist of a finite or infinite number of potential states (during action selection by the agent)? If the number of possible states is large, then the task envirionment is continuous, otherwise it is discrete.
Multi-Agent	Does the environment contain a single agent, or possibly multiple agents acting in a cooperative or competitive fashion.

ARCHITECTURE FOR INTELLIGENT AGENTS

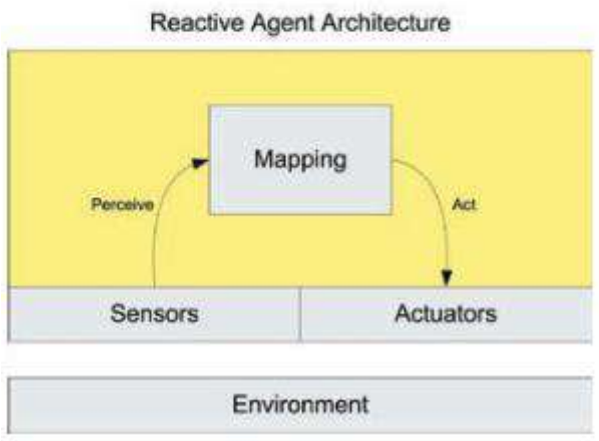
Agent architectures, like software architectures, are formally a description of the elements from which a system is built and the manner in which they communicate. Further, these elements can be defined from patterns with specific constraints.

A number of common architectures exist that go by the names *pipe-and- filter* or *layered* architecture. Note that these define the interconnections between components. Pipe-and-Filter defines a model where data is moved through a set of one or more objects that perform a transformation. Layered simply means that the system is comprised of a set of layers that provide a specific set of logical functionality and that connectivity is commonly restricted to the layers contiguous to one another.

Types of Architectures

Reactive Architectures
Deliberative Architectures
Blackboard Architectures
Belief-Desire-Intention (BDI) Architecture
Hybrid Architectures
Mobile Architectures

Reactive Architectures



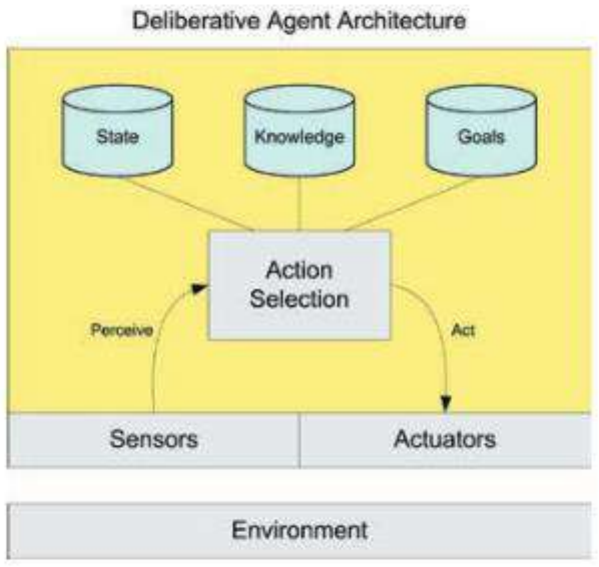
A reactive architecture is the simplest architecture for agents. In this architecture, agent behaviors are simply a mapping between stimulus and response. The agent has no decision-making skills, only reactions to the environment in which it exists.

Agent simply reads the environment and then maps the state of the environment to one or more actions. Given the environment, more than one action may be appropriate, and therefore the agent must choose.

Advantage
They are extremely fast.
This kind of architecture can be implemented easily in hardware, or fast in software lookup.

Disadvantage
They apply only to simple environments.
Sequences of actions require the presence of state, which is not encoded into the mapping function.

Deliberative Architectures



A deliberative architecture includes some deliberation over the action to perform given the current set of inputs.
It considers the sensors, state, prior results of given actions, and other information in order to select the best action to perform.
The mechanism for action selection could be a variety of mechanisms including a production system, neural network, or any other intelligent algorithm.

Advantage

It can be used to solve much more complex problems than the reactive architecture.

It can perform planning, and perform sequences of actions to achieve a goal.

Disadvantage

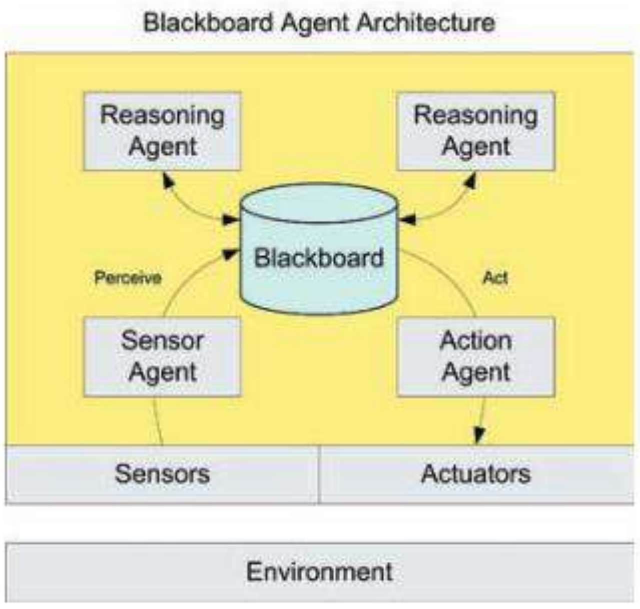
It is slower than the reactive architecture due to the deliberation for the action to select

Blackboard Architectures

The blackboard architecture is a very common architecture that is also very interesting. The first blackboard architecture was HEARSAY-II, which was a speech understanding system.

This architecture operates around a global work area call the *blackboard*.

The blackboard is a common work area for a number of agents that work cooperatively to solve a given problem. The blackboard therefore contains information about the environment, but also intermediate work results by the cooperative agents



In this example, two separate agents are used to sample the environment through the available sensors (the sensor agent) and also through the available actuators (action agent).

The blackboard contains the current state of the environment that is constantly updated by the sensor agent, and when an action can be performed (as specified in the blackboard), the action agent translates this action into control of the actuators.

The control of the agent system is provided by one or more reasoning agents. These agents work together to achieve the goals, which would also be contained in the blackboard. In this example, the first reasoning agent could implement the goal definition behaviors, where the second reasoning agent could implement the planning portion (to translate goals into sequences of actions).

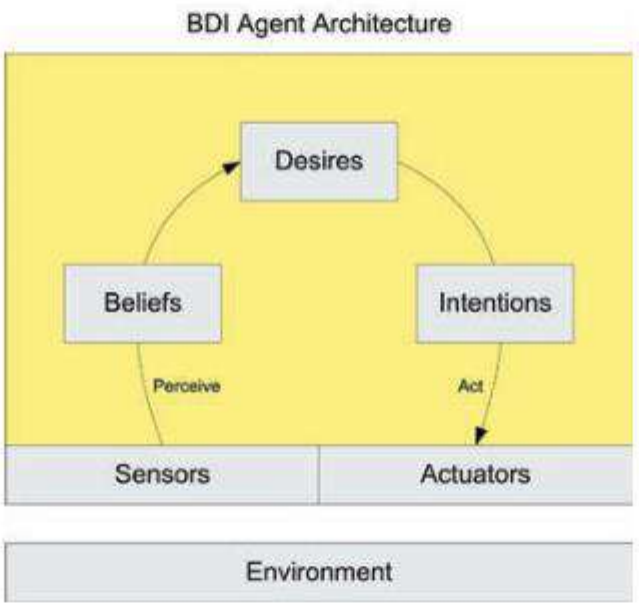
Since the blackboard is a common work area, *coordination must be provided such that agents* don't step over one another. For this reason, *agents are scheduled based on their need*. For example, agents can monitor the blackboard, and as information is added, they can request the ability to operate. The scheduler can then identify which agents desire to operate on the blackboard, and then invoke them accordingly.

The blackboard architecture, with its globally available work area, is easily implemented with a multi-threading system. Each agent becomes one or more system threads. From this perspective, the blackboard architecture is very common for agent and non-agent systems

Belief-Desire-Intention (BDI) Architecture

BDI, which stands for Belief-Desire-Intention, is an architecture that follows the theory of human reasoning as defined by Michael Bratman.

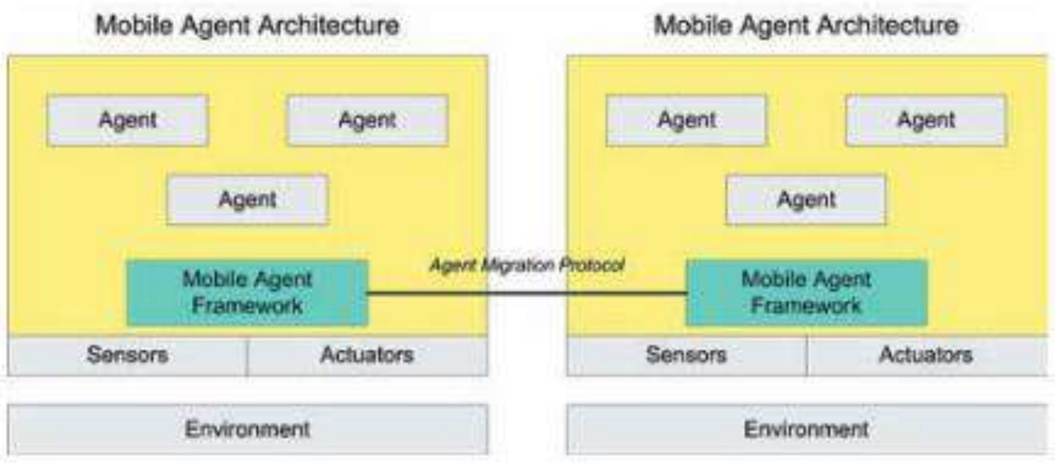
Belief represents the view of the world by the agent (what it believes to be the state of the environment in which it exists).
Desires are the goals that define the motivation of the agent (what it wants to achieve). The agent may have numerous desires, which must be consistent.
Intentions specify that the agent uses the Beliefs and Desires in order to choose one or more actions in order to meet the desires



BDI architecture defines the basic architecture of any deliberative agent. It stores a representation of the state of the environment (beliefs), maintains a set of goals (desires), and finally, an intentional element that maps desires to beliefs (to provide one or more actions that modify the state of the environment based on the agent's needs).

Hybrid Architectures
As is the case in traditional software architecture, most architectures are hybrids. For example, the architecture of a network stack is made up of a pipe-and-filter architecture and a layered architecture. This same stack also shares some elements of a blackboard architecture, as there are global elements that are visible and used by each component of the architecture.
The same is true for agent architectures. Based on the needs of the agent system, different architectural elements can be chosen to meet those needs.

Mobile Architectures
Mobile architectural pattern introduces the ability for agents to migrate themselves between hosts. The agent architecture includes the mobility element, which allows an agent to migrate from one host to another. An agent can migrate to any host that implements the mobile framework.



The mobile agent framework provides a protocol that permits communication between hosts for agent migration.
This framework also requires some kind of *authentication and security, to avoid a mobile agent framework from becoming a conduit for viruses*.
Also implicit in the mobile agent framework is a *means for discovery.* For example, which hosts are available for migration, and what services do they provide?
Communication is also implicit, as agents can communicate with one another on a host, or across hosts in preparation for migration.

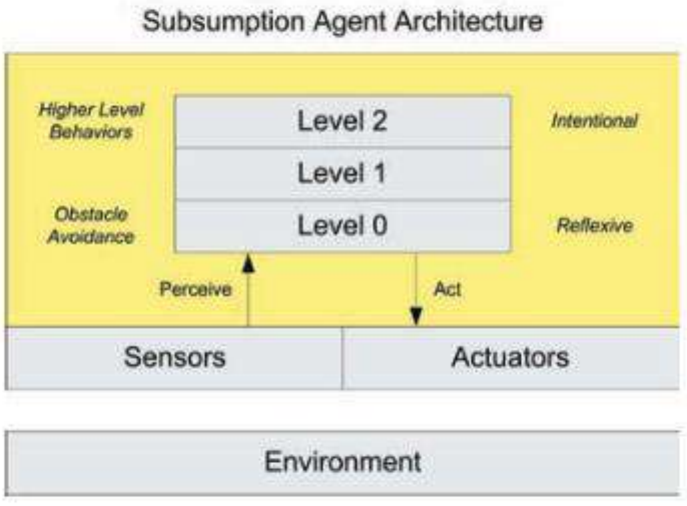
The mobile agent architecture is advantageous as it *supports the development of intelligent distributed systems*. But a distributed system that is dynamic, and whose configuration and loading is defined by the agents themselves.

Subsumption Architecture (Reactive Architecture)

The Subsumption architecture, originated by Rodney Brooks in the late 1980s, was created out of research in behavior-based robotics.

The fundamental idea behind subsumption is that intelligent behavior can be created through a collection of simple behavior modules. These behavior modules are collected into layers. At the bottom are behaviors that are reflexive in nature, and at the top, behaviors that are more complex.

At the bottom (level0) exist the reflexive behaviors (such as obstacle avoidance). If these behaviors are required, then level 0 consumes the inputs and provides an action at the output. But no obstacles exist, so the next layer up is permitted to subsume control. At each level, a set of behaviors with different goals compete for control based on the state of the environment.



To support this capability, levels can be inhibited (in other words, their outputs are disabled). Levels can also be suppressed such that sensor inputs are routed to higher layers.

The basic premise is that we begin with a simple set of behaviors, and once we've succeeded there, we extend with additional levels and higher-level behaviors. For example, we begin with obstacle avoidance and then extend for object seeking. From this perspective, the architecture takes a more evolutionary design approach.

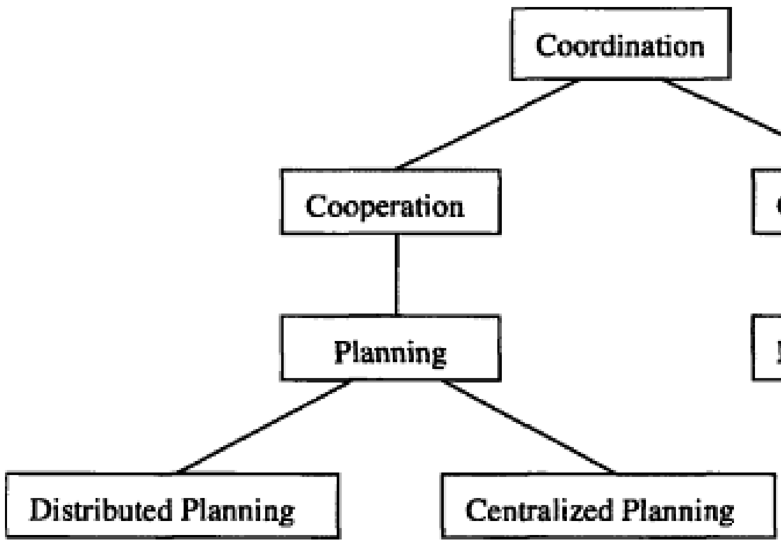
Subsumption does have its problems. It is simple, but it turns out not to be extremely extensible. As new layers are added, the layers tend to interfere with one another, and then the problem becomes how to layer the behaviors such that each has the opportunity to control when the time is right. Subsumption is also reactive in nature, meaning that in the end, the architecture still simply maps inputs to behaviors (no planning occurs, for example). What subsumption does provide is a means to choose which behavior for a given environment.

AGENT COMMUNICATION

An agent is an active object with the ability to perceive, reason, and act.

Agent has explicitly represented knowledge and a mechanism for operating on or drawing inferences from its knowledge.

Agent has the ability to communicate. This ability is part perception (the receiving of messages) and part action (the sending of messages).



TAXONAMY OF AGENT COMMUNICATION

A taxonomy of some of the different ways in which agents can coordinate their behavior and activities

Agents communicate in order to achieve better goals of themselves or of the society/system in which they exist.

Note that the goals might or might not be known to the agents explicitly, depending on whether or not the agents are goal- based.

Communication can enable the agents to coordinate their actions and behavior, resulting in systems that are more coherent.

Coordination

Coordination is a property of a system of agents performing some activity in a shared environment.

The degree of coordination is the extent to which they avoid extraneous activity by reducing resource contention, avoiding livelock and deadlock, and maintaining applicable safety conditions.

Cooperation is coordination among nonantagonistic agents, while negotiation is coordination among competitive or simply self-interested agents.

To cooperate successfully, each agent must maintain a model of the other agents, and also develop a model of future interactions. This presupposes sociability.

Coherence

Coherence is how well a system behaves as a unit. A problem for a multiagent system is how it can maintain global coherence without explicit global control.

The agents must be able on their own to determine goals they share with other agents, determine common tasks, avoid unnecessary conflicts, and pool knowledge and evidence. It is helpful if there is some form of organization among the agents.

Social commitments can be a means to achieving coherence

Dimensions of Meaning

Three aspects to the formal study of communication:

Syntax (how the symbols of communication are structured),

Semantics (what the symbols denote), and

Pragmatics (how the symbols are interpreted).

Meaning is a combination of semantics and pragmatics. Agents communicate in order to understand and be understood, so it is important to consider the different dimensions of meaning that are associated with communication

Descriptive vs. Prescriptive. Descriptions are important for human comprehension, but are difficult for agents to mimic. Most agent communication languages are designed for the exchange of information about activities and behavior.

Personal vs. Conventional Meaning.
An agent might have its **own meaning for a message**, but this might differ from the meaning conventionally accepted by the other agents with which the agent communicates.
Multiagent systems should opt for **conventional meanings,** especially since these systems are typically open environments in which new agents might be introduced at any time.

Subjective vs. Objective Meaning
A message often has an explicit effect on the environment, which can be perceived objectively. The effect might be different than that understood internally, i.e., subjectively, by the sender or receiver of the message.

Speaker's vs. Hearer's vs. Society's Perspective Independent of the conventional or objective meaning of a message, the message can be expressed according to the viewpoint of the speaker or hearer or other observers.

Semantics vs. Pragmatics The pragmatics of a communication are concerned with how the communicators use the communication. This includes considerationsof the mental states of the communicators and the environment in which they exist, considerations that are external to the syntax and semantics of the communication.

Contextuality Messages cannot be understood in isolation, but must be interpreted in terms of the mental states of the agents, the present state of the environment, and the environment's history: how it arrived at its present state. Interpretations are directly affected by previous messages and actions of the agents.

Coverage Smaller languages are more manageable, but they must be large enough so that an agent can convey the meanings it intends.

Identity When a communication occurs among agents, its meaning is dependent on the identities and roles of the agents involved, and on how the involved agents are specified. A message might be sent to a particular agent, or to just any agent satisfying a specified criterion.

Cardinality A message sent privately to one agent would be understood differently than the same message broadcast publicly

Message Types
Two basic message types:
Assertions
Queries
Assertion
Every agent, whether active or passive, must have the ability to accept information. Information is communicated to the agent from an external source by means of an assertion.
Queries
In order to assume a *passive role* in a dialog, an agent must additionally be able to answer questions, i.e., it must be able to

1) accept a query from an external source and
2) send a reply to the source by making an assertion.

In order to assume an *active role* in a dialog, an *agent must be able to issue queries and make assertions*. With these capabilities, the agent then can potentially control another agent by causing it to respond to the query or to accept the information asserted. This means of control can be extended to the control of subagents, such as neural networks and databases.

An agent functioning as a peer with another agent can assume both active and passive roles in a dialog. It must be able to make and accept both assertions and queries

Communication Levels
Communication protocols are typically specified at several levels.
Lowest level of the protocol specifies the **method of interconnection**;
Middle level **specifies the format, or syntax, of the information being transfered**;
Top level **specifies the meaning, or semantics, of the information**. The semantics refers not only to the substance of the message, but also to the type of the message.

There are both *binary and n-ary communication protocols*.
A binary protocol involves a single sender and a single receiver, whereas an n-ary protocol involves a single sender and multiple receivers (sometimes called broadcast or multicast).

A protocol is specified by a **data structure with the following five fields**:
1. sender
2. receiver(s)
3. language in the protocol
4. encoding and decoding functions
5. actions to be taken by the receiver(s)

Speech Acts
Spoken human communication is used as the model for communication among computational agents. Speech act theory views human natural language as *actions*, such as requests, suggestions, commitments, and replies.
A speech act has three aspects:
1. **Locution**, the physical *utterance* by the speaker
2. **Illocution**, the *intended meaning* of the utterance by the speaker
3. **Perlocution**, the *action* that results from the locution

In communication among humans, the intent of the message is not always easily identified. For example, "I am cold," can be viewed as an assertion, a request for a sweater, or a demand for an increase in room temperature. However, for communication among agents, we want to insure that there is no doubt about the type of message.

Speech act theory helps *define the type of message* by using the concept of the illocutionary force, which constrains the semantics of the communication act itself.
The sender's intended communication act is clearly defined, and the receiver has no doubt as to the type of message sent. This constraint simplifies the design of our software agents. The message contained *within* the protocol may be ambiguous, may have no simple response, or may require decomposition and the assistance of other agents; however, the communication protocol itself should clearly identify the type of message being sent.

Knowledge Query and Manipulation Language (KQML)

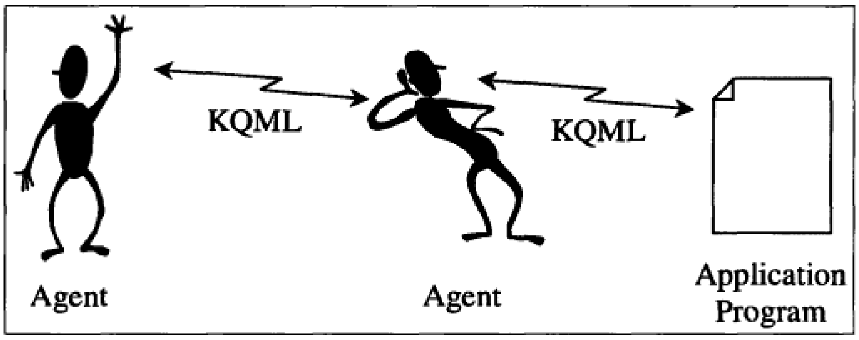
A fundamental decision for the interaction of agents is to separate the semantics of the communication protocol (which must be domain independent) from the semantics of the enclosed message (which may depend on the domain).

The communication *protocol must be universally shared by all agents*. It should be *concise and have only a limited number of primitive* communication acts.

The knowledge query and manipulation language (KQML) is a protocol for exchanging information and knowledge.

The elegance of KQML is that all information for understanding the content of the message is included in the communication itself. The basic protocol is defined by the following structure:

```
(KQML-performative
:sender          <word>
:receiver        <word>
:language        <word>
:ontology        <word>
:content         <expression>
...)
```



KQML is a protocol for communications among both agents and application programs.

The KQML-performatives are modeled on speech act performatives. Thus, the semantics of KQML performatives is domain independent, while the semanatics of the message is defined by the fields

:content (the message itself),
:language (the langauge in which the message is expressed), and
:ontology (the vocabulary of the "words" in the message).

In effect, KQML "wraps" a message in a *structure* that can be understood by any agent. (To understand the message itself, the recipient must understand the language and have access to the ontology.)

The terms :content, :language, and :ontology delineate the semantics of the message.
Other arguments, including :sender, :receiver, :reply-with, and :in-reply-to, are parameters of the message passing.

KQML assumes asynchronous communications; the fields: reply-with from a sender and: in-reply-to from a responding agent *link an outgoing message with an expected response*.

KQML is part of a broad research effort to develop a methodology for distributing information among different systems.

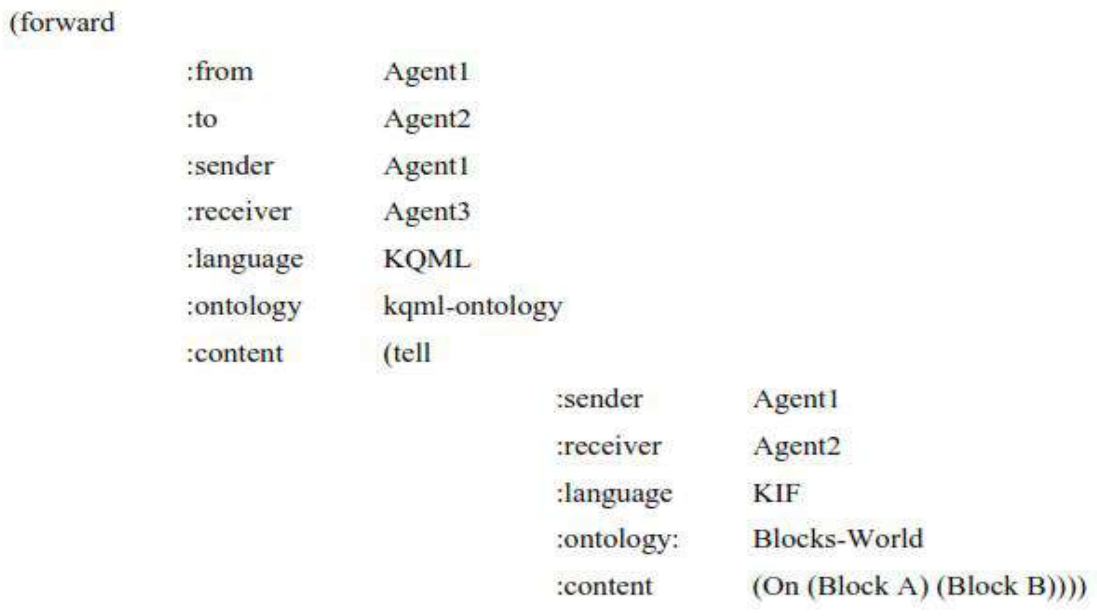
One part of the effort involves defining the **Knowledge Interchange Format (KIF)**, a formal syntax for representing knowledge KIF is largely based on first-order predicate calculus.
Another part of the effort is defining ontologies that define the common concepts, attributes, and relationships for different subsets of world knowledge.
The definitions of the ontology terms give meaning to expressions represented in KIF.

For example, in a Blocks-World ontology, if the concept of a wooden block of a given size is represented by the unary predicate Block, then the fact that block A is on top of block B could be communicated as follows:

```
(tell
:sender         Agent1
:receiver       Agent2
:language:      KIF
:ontology:      Blocks-World
:content        (AND (Block A) (Block B) (On A B))
```

The language in a KQML message is not restricted to KIF; other languages such as PROLOG, LISP, SQL, or any other defined agent communication language can be used.

The KQML performatives may be organized into seven basic categories:
• **Basic query** performatives (evaluate, ask-one, ask-all, ...)
• **Multiresponse query** performatives (stream-in, stream-all, ...)
• **Response** performatives (reply, sorry, ...)
• **Generic informational** performatives (tell, achieve, cancel, untell, unachieve, ...)
• **Generator** performatives (standby, ready, next, rest, ...)
• **Capability-definition** performatives (advertise, subscribe, monitor, ...)
• **Networking** performatives (register, unregister, forward, broadcast, ...)
KQML-speaking agents appear to each other as clients and servers. Their communications can be either synchronous or asynchronous
KQML messages can be "nested" in that the content of a KQML message may be another KQML message, which is self contained. For example, if Agent1 cannot communicate directly with Agent2 (but can communicate with Agent3), Agent1 might ask Agent3 to forward a message to Agent2:



```
(forward
        :from           Agent1
        :to             Agent2
        :sender         Agent1
        :receiver       Agent3
        :language       KQML
        :ontology       kqml-ontology
        :content        (tell
                                :sender         Agent1
                                :receiver       Agent2
                                :language       KIF
                                :ontology:      Blocks-World
                                :content        (On (Block A) (Block B))))
```

NEGOTIATION
A frequent form of interaction that occurs among agents with different goals is termed negotiation.
Negotiation is a process by which a joint decision is reached by two or more agents, each trying to reach an individual goal or objective.

The agents first communicate their positions, which might conflict, and then try to move towards agreement by making concessions or searching for alternatives.

The major features of negotiation are
(1) **Language** used by the participating agents,
(2) **Protocol** followed by the agents as they negotiate, and
(3) **Decision process** that each agent uses to determine its positions, concessions, and criteria for agreement.
Two types of systems and techniques for negotiation:
Environment-centered
Agent centered.

Environment-centered techniques
Developers of environment-centered techniques focus on the following problem:
"How can the rules of the environment be designed so that the agents in it, regardless of their origin, capabilities, or intentions, will interact productively and fairly?"

The resultant negotiation mechanism should ideally have the following **attributes**
Efficiency: the agents should not waste resources in coming to an agreement. Stability: no agent should have anincentive to deviate from agreed-upon strategies.
Simplicity: the negotiation mechanism should impose low computational and bandwidth demands on the agents.
Distribution: the mechanism should not require a central decision maker.
Symmetry: the mechanism should not be biased against any agent for arbitrary or inappropriate reasons.

Three types of environments have been identified:
Worth-oriented domains,
State-oriented domains,
Task-oriented domains.
A task-oriented domain is one where agents have a set of tasks to achieve, all resources needed to achieve the tasks are available, and the agents can achieve the tasks without help or interference from each other. However, the agents can benefit by sharing some of the tasks.

Example:
An example is the "Internet downloading domain", where each agent is given a list of documents that it must access over the Internet. There is a cost associated with downloading, which each agent would like to minimize. If a document is common to several agents, then they can save downloading cost by accessing the document once and then sharing it.

The environment might provide the following simple negotiation mechanism and constraints:
(1) Each agent declares the documents it wants,
(2) Documents found to be common to two or more agents are assigned to agents based on the toss of a coin,
(3) Agents pay for the documents they download, and
(4) Agents are granted access to the documents they download as well as any in their common sets.

This mechanism is simple, symmetric, distributed, and efficient (no document is downloaded twice). To determine stability, the agents' strategies must be considered.

Agents' Strategies
An optimal strategy is for an agent to declare the true set of documents that it needs, regardless of what strategy the other agents adopt or the documents they need. Because there is no incentive for an agent to diverge from this strategy, it is stable.

Agent-centered negotiation mechanisms
Developers of agent-centered negotiation mechanisms focus on the following problem: "Given an environment in which my agent must operate, what is the best strategy for it to follow?"

Most such negotiation strategies have been developed for specific problems, so few general principles of negotiation have emerged.
Two general approaches:

Speech-act classifiers with a possible world semantics
For the first approach, **speech-act classifiers together with a possible world semantics** are used to formalize negotiation protocols and their components. *This clarifies the conditions of satisfaction for different kinds of messages*.

$$\forall x (x \neq y) \land$$
$$\neg(\mathit{Precommit}_a\ y\ x\ \phi) \land (\mathit{Goal}\ y\ \mathit{Eventually}(\mathit{Achieves}\ y\ \phi)) \land (\mathit{Willing}\ y\ \phi)$$
$$\Longleftrightarrow (\mathit{Intend}\ y\ \mathit{Eventually}(\mathit{Achieves}\ y\ \phi))$$

This rule states that an agent forms and maintains its commitment to achieve ø individually iff (1) it has not pre-committed itself to another agent to adopt and achieve ø, (2) it has a goal to achieve ø individually, and (3) it is willing to achieve ø individually.

Agents are economically rational
The **second approach** is **based on an assumption that the agents are economically rational**. Further, the set of agents must be small, they must have a common language and common problem abstraction, and they must reach a common solution. Under these assumptions, Rosenschein and Zlotkin developed a unified negotiation protocol.

Unified negotiation protocol:
Agents create a deal, that is, a joint plan between the agents that would satisfy all of their goals.
The *utility of a deal for an agent* is the amount he is willing to pay minus the cost of the deal.
Each *agent wants to maximize* its own utility.
The agents discuss a negotiation set, which is the set of all deals that have a positive utility for every agent.
In formal terms, a task-oriented domain under this approach becomes a tuple <T, A, c>
where T is the set of tasks, A is the set of agents, and c(X) is a monotonic function for the cost of executing the tasks X.
A deal is a redistribution of tasks. The utility of deal d for agent k is Uk(d) = c(Tk) - c(dk)
The conflict deal D occurs when the agents cannot reach a deal.
A deal d is individually rational if d > D.
Deal d is pareto optimal if there is no deal d' > d.
The set of all deals that are individually rational and pareto optimal is the negotiation set, NS.

There are three possible situations:
1. **Conflict**: the negotiation set is empty
2. **Compromise**: agents prefer to be alone, but since they are not, they will agree to a negotiated deal
3. **Cooperative**: all deals in the negotiation set are preferred by both agents over achieving their goals alone.
When there is a conflict, then the agents will not benefit by negotiating—they are better off acting alone. Alternatively, they can "flip a coin" to decide which agent gets to satisfy its goals. Negotiation is the best alternative in the other two cases.

Since the agents have some *execution autonomy*, they can in principle deceive or mislead each other. Therefore, an interesting research problem is to develop protocols or societies in which the effects of deception and misinformation can be constrained. Another aspect of the research problem is to develop protocols under which it is rational for agents to be honest with each other.

BARGAINING
In a bargaining setting, **agents can make a mutually beneficial agreement**, but have a conflict of interest about which agreement to make.

A monopolist gets all of the gains from interaction while an agent facing perfect competition can make no profit. Real world settings usually *consist of a finite number of competing agents, so neither monopoly nor perfect competition assumptions strictly apply*. Bargaining theory fits in this gap.

There are two major subfields of bargaining theory: axiomatic and strategic.

Axiomatic Bargaining Theory
Axiomatic bargaining theory uses strategies form of equilibrium.
Desirable properties for a solution, called axioms of the bargaining solution, are postulated
Solution concept that satisfies these axioms is sought

2-agent setting where the agents have to decide on an outcome $o \in O$,
Decide fallback outcome $O_{fallback}$. occurs if no agreement is reached.

There is a utility function :$u_i : O \rightarrow \Re$ for each agent $i \in$ [1,2].

It is assumed that the set of feasible utility vectors $\{(u_1(o), u_2(o)) | o \in O\}$ is convex. This occurs, for example, if outcomes include all possible lotteries over actual alternatives.

When many deals are individually rational—i.e. have higher utility than the fallback—to both agents, multiple Nash equilibria often exist.

For example, if the agents are bargaining over how to split a dollar, all splits that give each agent
more than zero are in equilibrium.

If agent one's strategy is to offer p and no more, agent two's best response is to take the offer as opposed to the fallback which is zero.

Now, one's best response to this is to offer p and no more. Thus, a Nash equilibrium exists for any p that defines a contract that is individually rational for both agents, and feasible (0 <<
1). Due to the nonuniqueness of the equilibrium, a stronger (axiomatic) solution concept such as the Nash bargaining solution is needed to prescribe a unique solution.

The axioms of the Nash bargaining solution $u^* = (u_1\ (o^*), u_2\ (o^*))$ are:

Invariance: The agents' numeric utility functions really only represent ordinal preferences among outcomes—the actual cardinalities of the utilities do not matter. Therefore, it should be possible to transform the utility functions in tile following way: for any strictly increasing linear function f, $u^*(f(o), f(O_{fallback})) = f(u^*(o, O_{fallback}))$.
- **Anonymity** (symmetry): switching labels on the players does not affect the outcome.
- **Independence of irrelevant alternatives**: if some outcomes o axe removed, but o^* is not, then o^* still remains the solution.
- **Pareto efficiency**: it is not feasible to give both players higher utility than under $u^* = (u_1(o^*), u_2(o^*))$.

Theorem 5.8 Nash bargaining solution

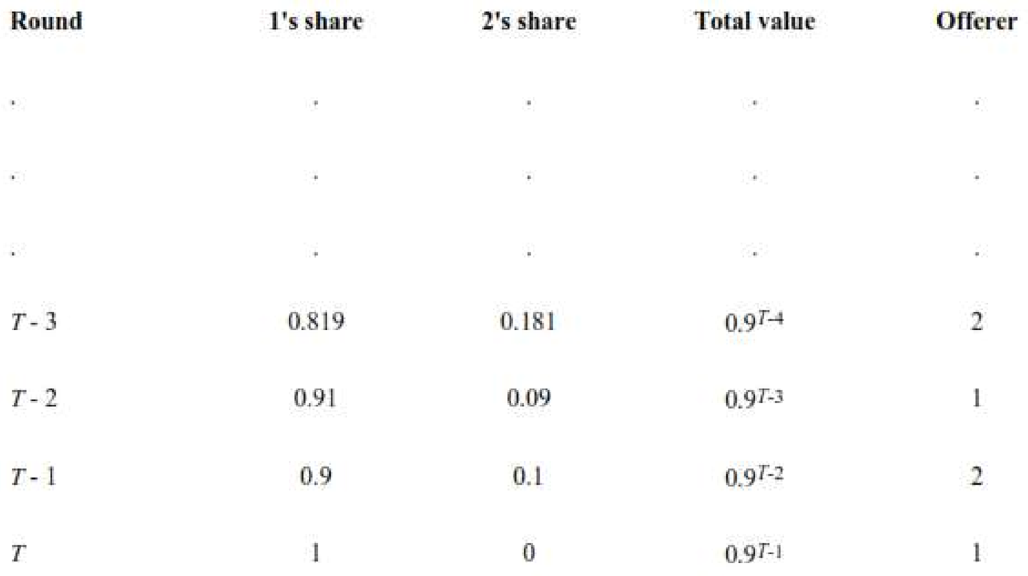
The unique solution that satisfies these four axioms is:

$$o^* = \arg\max_o [u_1(o) - u_1(o_{fallback})][u_2(o) - u_2(o_{fallback})]$$

The Nash bargaining solution can be directly extended to more than two agents, as long as the fallback occurs if at least one agent disagrees. The 2-agent Nash bargaining solution is also the 2-agent special case of the Shapley value—a particular solution concept for payoff division in coalition formation, discussed later in Section 5.8.3—where coalitions of agents can cooperate even if all agents do not agree.

<u>Strategic Bargaining Theory</u>

strategic bargaining theory, bargaining situation is modeled as a game, and the solution concept is based on an analysis of which of the players' strategies are in equilibrium.
Strategic bargaining theory explains the behavior of rational utility maximizing agents based on what the agents can choose for strategies, imposed notions of fairness.



Round	1's share	2's share	Total value	Offerer
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
T - 3	0.819	0.181	0.9^{T-4}	2
T - 2	0.91	0.09	0.9^{T-3}	1
T - 1	0.9	0.1	0.9^{T-2}	2
T	1	0	0.9^{T-1}	1

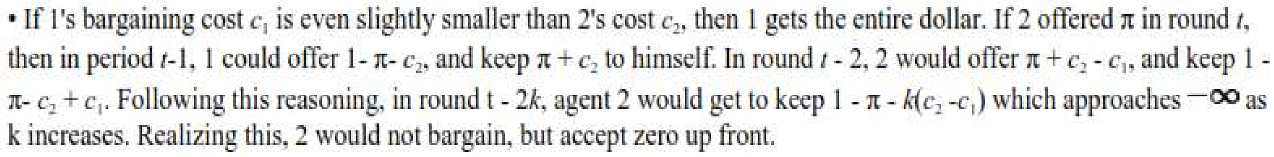
Table 5.4 Offerer's maximal acceptable claims in a finite game.

Strategic bargaining theory usually analyses sequential bargaining where agents alternate in making offers to each other in a prespecified order [50, 54, 51, 35]. Agent 1 gets to make the first offer. As an example, one can again think of deciding how to split a dollar. In a protocol with a finite number of offers and no time discount, the unique payoffs of the subgame perfect Nash equilibria are such that the last offerer will get the whole dollar (minus), because the other agent is better off accepting e than by rejecting and receiving nothing.

Sequential bargaining

Another model of sequential bargaining does not use discounts, but assumes a fixed bargaining cost per negotiation round.

• If the agents have *symmetric bargaining costs*, the solution concept is again powerless because any split of the dollar can be supported in subgame perfect Nash equilibrium.



• If 1's bargaining cost c_1 is even slightly smaller than 2's cost c_2, then 1 gets the entire dollar. If 2 offered π in round t, then in period t-1, 1 could offer 1- π- c_2, and keep $\pi + c_2$ to himself. In round t - 2, 2 would offer $\pi + c_2 - c_1$, and keep 1 - π- $c_2 + c_1$. Following this reasoning, in round t - 2k, agent 2 would get to keep 1 - π - $k(c_2 - c_1)$ which approaches $-\infty$ as k increases. Realizing this, 2 would not bargain, but accept zero up front.

• If 1's bargaining cost is greater than 2's, then 1 receives a payoff that equals the second agent's bargaining cost, and agent 2 receives the rest. Agreement is again reached on the first round. This case is equivalent to the previous case except that the agent with the smaller bargaining cost is willing to give the other agent c_2 in order to avoid going through the first period of bargaining.

Computation in Bargaining

All of the bargaining models discussed above *assume perfect rationality from the agents*. No computation is required in finding a mutually desirable contract. The space of deals is assumed to be fully comprehended by the agents, and the value of each potential contract known.

On the other hand, future work should focus on developing methods- where the cost of search (deliberation) for solutions is explicit, and it is decision-theoretically traded off against the bargaining gains that the search provides. This becomes particularly important as the bargaining techniques are scaled up to **combinatorial problems** with a multidimensional negotiation space as opposed to combinatorially simple ones like splitting the dollar.

There are actually *two searches occurring in bargaining*.

In the ***intra-agent deliberative search**, an* agent locally generates alternatives, evaluates them, counter speculates, does look ahead in the negotiation process *etc*.

In the ***inter-agent committal search***, the agents make (binding) agreements with each other regarding the solution. The agreements may occur over one part of the solution at a time.

The agreed issues provide context for more focused intra-agent deliberative search—thus *reducing the amount of costly computation required*.
The committal search may also *involve iteratively renegotiating* some parts of the solution that have already been agreed on, but have become less desirable in light of the newer agreements regarding other parts of the solution.

The two-search model proposed here is similar to the Real-Time A* search where an agent has to trade off thorough deliberation against more real-world actions. Similarly, in modeling bargaining settings that require nontrivial computations, each agent's strategy should incorporate both negotiation actions and deliberation actions. The bargaining setting is more complex than the single agent setting of Real-Time A* in that there are multiple self-interested agents: the agents' strategies should be in equilibrium.

ARGUMENTATION AMONG AGENTS

Argumentation theory with AI offers argumentation theory a laboratory for examining implementations of its rules and concepts.
Approaches to argumentation in AI integrate insights from different perspectives. In the artificial systems perspective, the aim is to build computer programs that model or support argumentative tasks.

Argumentation theory, or **argumentation**, is the interdisciplinary study of how conclusions can be reached through logical reasoning; that is, claims based, soundly or not, on premises. It includes the arts and sciences of civil debate, dialogue, conversation, and persuasion. It studies rules of inference, logic, and procedural rules in both artificial and real world settings

Argumentation includes deliberation and negotiation which are concerned with collaborative decision-making procedures.

Argumentation is one of four rhetorical modes (also known as *modes of discourse*), along with exposition, description, and narration.

Key components of argumentation

Understanding and identifying arguments
Identifying the premises from which conclusions are derived
Establishing the "burden of proof" – determining who made the initial claim and is thus responsible for providing evidence why his/her position merits acceptance.
For the one carrying the "burden of proof", the advocate, to marshal evidence for his/her position in order to convince or force the opponent's acceptance.
In a debate, fulfillment of the burden of proof creates a burden of rejoinder. One must try to identify faulty reasoning in the opponent's argument, to attack the reasons/premises of the argument, to provide counterexamples if possible, to identify any fallacies, and to show why a valid conclusion cannot be derived from the reasons provided for his/her argument

Internal structure of arguments

Typically an argument has an internal structure, comprising the following
a set of **assumptions** or premises
a method of **reasoning** or deduction and
a **conclusion** or point.
An argument has one or more premises and one conclusion.

Often classical logic is used as the method of reasoning so that the conclusion follows logically from the assumptions or support. One challenge is that if the set of assumptions is inconsistent then anything can follow logically from inconsistency.
In its most common form, argumentation involves an individual and an interlocutor or opponent engaged in dialogue, *each contending differing positions and trying to persuade each other*. Other types of dialogue in addition to persuasion are eristic, information seeking, inquiry, negotiation, deliberation, and the dialectical method

Toulmin Model of Argument
In *The Uses of Argument* (1958), Toulmin introduced what became known as the **Toulmin Model of Argument,** which broke argument into **six interrelated components**:
Claim: Conclusions whose merit must be established. For example, if a person tries to convince a listener that he is a British citizen, the claim would be "I am a British citizen." (1)
Data: The facts we appeal to as a foundation for the claim. For example, the person introduced in 1 can support his claim with the supporting data "I was born in Bermuda." (2)
Warrant: The statement authorizing our movement from the data to the claim. In order to move from the data established in 2, "I was born in Bermuda," to the claim in 1, "I am a British citizen," the person must supply a warrant to bridge the gap between 1 & 2 with the statement "A man born in Bermuda will legally be a British Citizen."
Backing: Credentials *designed to certify* the statement expressed in the warrant; backing must be introduced when the warrant itself is not convincing enough to the readers or the listeners.
For example, if the listener does not deem the warrant in 3 as credible, the speaker will supply the legal provisions as backing statement to show that it is true that "A man born in Bermuda will legally be a British Citizen."
Rebuttal: Statements recognizing *the restrictions to which the claim may legitimately be applied*. The rebuttal is exemplified as follows, "A man born in Bermuda will legally be a British citizen, unless he has betrayed Britain and has become a spy of another country."
Qualifier: Words or phrases expressing the speaker's degree of force or certainty concerning the claim. Such words or phrases include "possible," "probably," "impossible," "certainly," "presumably," "as far as the evidence goes," or "necessarily." The claim "I am definitely a British citizen" has a greater degree of force than the claim "I am a British citizen, presumably."
The first three elements "claim", "data", and "warrant" are considered as the essential components of practical arguments, while the second triad "qualifier", "backing", and "rebuttal" may not be needed in some arguments.
Two kinds of defeaters:
"rebutting" and "undercutting defeaters."
A defeater is rebutting, when it is a reason for the opposite conclusion (Fig. 11.3, left).
Undercutting defeaters attack the connection between the reason and the conclusion and not the conclusion itself (Fig. 11.3, right). The example about looking red concerns an undercutting defeater

- **Fig. 11.2** Pollock's red light example

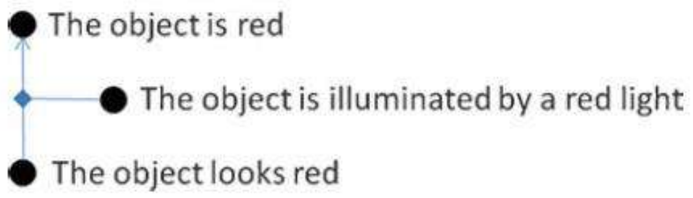
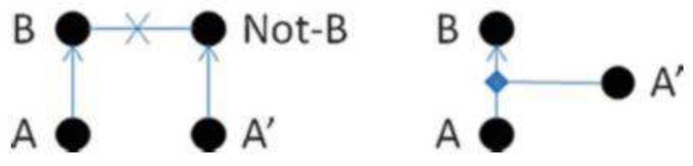


Fig. 11.3 A rebutting defeater and an undercutting defeater



A rebutting defeater and an undercutting defeater

The example about looking red concerns an undercutting defeater since when there is a red light, it is not attacked that the object is red, but merely that the object's looking red is a reason for its being red. A defeater is rebutting, when it is a reason for the opposite conclusion.

List of important classes of specific reasons:

1. **Deductive reasons**. These are the **conclusive reasons** as they are in particular studied in standard classical logic. For instance, P^Q is a reason for P and for Q, and P and Q taken together are a reason for P^Q.
2. **Perception**. When we **perceive our world**, the resulting perceptual state provides us with prima facie reasons pertaining to our world. Pollock says that no
3. **Memory**. **Justified beliefs can also be arrived at by reasoning**. The results of reasoning can be rejected when a belief used in the reasoning is later rejected by us
4. **Statistical syllogism**. Pollock describes the statistical syllogism as the simplest form of probabilistic reasoning: from **"Most F's are G" and "This is an F'," we can conclude prima facie "This is a G."** The strength of the reason depends on the probability of F's being G.
5. **Induction**. Pollock discusses two kinds of induction: (a) in enumerative induc- tion, we conclude that all F's are G when all F's observed until now have been G; (b) in statistical induction, we conclude that the proportion of F's being G is approximately r when the proportion of observed F's being G is r.

Forms of Argument Defeat

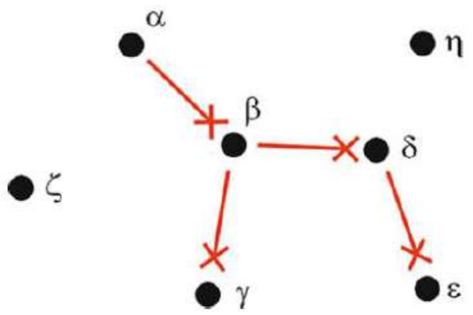
1. An argument can be **undermined**. In this form of defeat, the premises or assumptions of an argument are attacked. This form of defeat corresponds to Hart's denial of the premises.
2. An argument can be **undercut**. In this form of defeat, the connection between a
(set of) reason(s) and a conclusion in an argument is attacked.
3. An argument can be **rebutted**. In this form of defeat, an argument is attacked by giving an argument for an opposite conclusion.

An example is an argument based on the sorites paradox:

This body of grains of sand is a heap.
So, this body of grains of sand minus 1 grain is a heap.
So, this body of grains of sand minus 2 grains is a heap.
. . .
So, this body of grains of sand minus n grains is a heap.

At some point, the argument breaks down, in particular when n exceeds the total amount of grains of sand to start with.

Abstract Argumentation

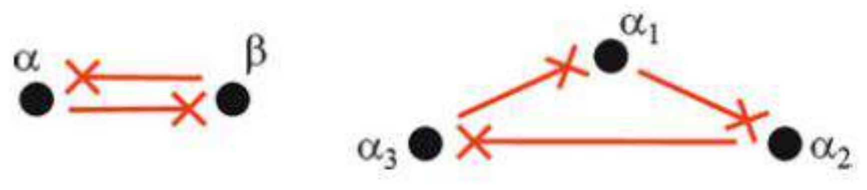


Argumentation frame- work

The formal study of the attack relation between arguments, thereby separating the properties depending exclusively on argument attack from any concerns related to the structure of the arguments. Mathematically speaking, the argument attack relation is a directed graph, the nodes of which are the arguments, whereas the edges represent that one argument attacks another. Such a directed graph is called an argumentation frame-work. Above Figure shows an example of an argumentation framework, with the dots representing arguments and the arrows (ending in a cross to emphasize the attacking nature of the connection) representing argument attack. In above Fig., the argument α attacks the argument β, which in turn attacks both γ and δ.

A *set of arguments is admissible if two conditions obtain*:
1. The set of arguments is conflict-free, i.e., does not contain an argument that attacks another argument in the set.
2. Each argument in the set is acceptable with respect to the set, i.e., when an argument in the set is attacked by another argument (which by (1) cannot be in the set itself), the set contains an argument that attacks the attacker.

Arguments attacking each other in cycles



The cycle of attacks on the right containing three arguments, α1, α2, and α3, is an example of the second fact above, the fact that it can happen that an argument is neither admissibly provable nor refutable. This follows from the fact that there is no admissible set that contains (at least) one of the arguments, α1, α2, or α3. Suppose that the argument α3 is in an admissible set. Then the set should defend α3 against the argument α2, which attacks α3. This means that α1 should also be in the set, since it is the only argument that can defend α3 against α2. But this is not possible, because then α1 and α3 are both in the set, introducing a conflict in the set. As a result, there is only one admissible set: the empty set that contains no arguments at all.

Argument Schemes
Argument schemes can be thought of as analogues of the rules of inference of classical logic. An example of a rule of inference is, for instance, the following version of modus ponens:

P
If P, then Q
Therefore: Q

Whereas logical rules of inference, such as modus ponens, are abstract, strict, and (usually) considered to have universal validity, argument schemes are concrete, defeasible, and context dependent. An example is the following scheme for witness testimony:

Witness A has testified that P.
Therefore: P

TRUST AND REPUTATION IN MULTI-AGENT SYSTEMS
Trust is subjective and contingent on the uncertainty of future outcome (as a result of trusting).
It depends on the level we apply it:
- **User confidence**
• Can we trust the user behind the agent?
– Is he/she a trustworthy source of some kind of knowledge? (e.g. an expert in field)
– Does he/she acts in the agent system (through his agents in a trustworthy way?

- Trust of users in agents
- Issues of autonomy: the more autonomy, less trust
- How to create trust?
– Reliability testing for agents
– Formal methods for open MAS
– Security and verifiability
- Trust of agents in agents
- Reputation mechanisms
- Contracts
- Norms and Social Structures

Why Trust?

Closed environments

- In **closed environments**, cooperation among agents is included as part of the designing process:
- The multi-agent system is usually built by a single developer or a single team of developers and the chosen developer or a single team of developers, and the chosen option to reduce complexity is to ensure cooperation among the agents they build including it as an important system requirement.
- **Benevolence assumption**: an agent ai requesting information or a certain service from agent aj can be sure that such agent will answer him if aj has the capabilities and the resources needed, otherwise aj will inform ai that it cannot perform the action requested.
- It can be said that in closed environments trust is implicit.

Open environment trust

However, in an open environment trust is not easy to achieve, as

- Agents introduced by the system designer can be expected to be nice trustworthy, this be ensured for alien agents out of the designer control
- These alien agents may give incomplete or false information to other agents or betray them if such actions allow them to fulfill their individual goals.
- In such scenarios developers use to create competitive systems where each agent seeks to maximize its own expected utility at agents.
- But, what if solutions can only be constructed by means of cooperative problem solving?
- Agents should try to cooperate, even if there is some uncertainty about the other agent's behaviour
- That is, to have some explicit representation of trust

Computing trust

- Trust value can be assigned to an agent or to a group of agents
- Trust value is an asymmetrical function between agent a1 and a2
- trust_val(a1,a2) does not need to be equal to trust_val(a2,a1)
- Trust can be computed as

A binary value (1='I do trust this agent', 0='I don't trust this agent')

A set of qualitative values or a discrete set of numerical values

(e.g. trust always , trust conditional to X , no trust) (e.g. '2', '1', '0', '-1', '-2') 'trust always' ' ' 'no trust'

A continuous numerical value (e.g. [-300..300])

A probability distribution

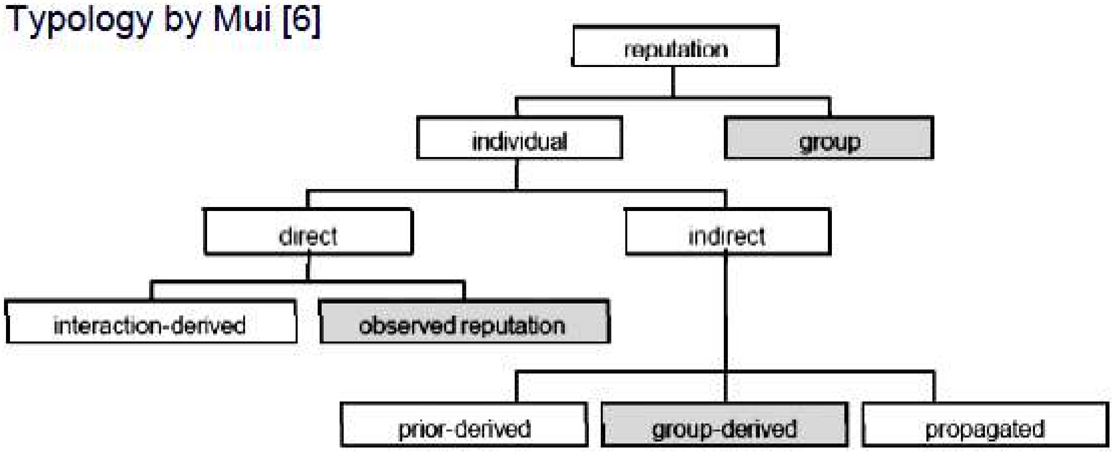
Degrees over underlying beliefs and intentions (cognitive approach)

- Trust values can be **externally defined by the system designer**: the trust values are pre-defined
- **By the human user**: he can introduce his trust values about the humans behind the agents
- Trust values **can be inferred from some existing representation** about the interrelations between the agents
- Communication patterns, cooperation history logs, e-mails,webpage connectivity mapping…
- Trust values **can be learnt from current and past experiences**
- Increase trust value for agent ai if behaves properly with us
- Decrease trust value for agent ai if it fails/defects us
- Trust values can be propagated or shared through a MAS
- Recommender systems, Reputation mechanisms

Trust and Reputation

- Most authors in literature make a mix between trust and reputation
- Some authors make a distinction between them
- **Trust** is an individual measure of confidence that a given gent has over other gent(s)
- **Reputation** is a social measure of confidence that a group of agents or a society has over agents or groups
- (social) Reputation is one mechanism to compute (individual) Trust
- I will trust more an agent that has good reputation
- My reputation clearly affects the amount of trust that others have towards me.
- Reputation can have a ***sanctioning*** role in social groups: a bad reputation can be very costly to one's future transactions.
- Most authors combine (individual) Trust with some form of (social) Reputation in their models

Typology by Mui [6]



- At the topmost level, reputation can be used to describe an individual or a group of individuals
- The most typical in reputation systems is the individual reputation
- Group reputation is the reputation of a set of agents
- E.g., a team, a firm, a company
- Group reputation can help compute the reputation of an individual.
- E.g., Mr. Anderson worked for Google Labs in Palo Alto.

***Direct experiences are the most relevant and reliable information source for individual* trust/reputation**

□Type 1: Experience based on *direct interaction* with the partner

□ Used by almost all models

□ How to:

- trust value about that partner increases with good experiences,
- it decreases with bad ones

□ Problem: how to compute trust if there is no previous interaction?

Type 2: Experience based on *observed interaction* of other members

□ *Used only in scenarios prepared for this*.

□ How to: depends on what an agent can observe

a) agents can access to the log of past interactions of other agents

b) agents can access some feedback from agents about their past interactions (e.g., in eBay)

□ Problem: one has to introduce some noise handling or confidence level on this information

Indirect experiences as source

Prior-derived:

agents bring with them prior beliefs about strangers

□ Used by some models to initialize trust/reputation values

□ How-to:

a) designer or human user assigns prior values

b) a uniform distribution for reputation priors is set

c) give new agents the lowest possible reputation value

•there is no incentive to throw away a cyber identity when an agent's reputation falls below a starting point.

d) assume neither good nor bad reputation for unknown agents.

•Avoid lowest reputation for new, valid agents as an obstacle for other agents to realise that they are valid.

□Problem: prior beliefs are common in human societies (sexual or racial prejudices), but hard to set in software agents

Group-derived:

models for groups can been extended to provide prior reputation estimates for agents in social groups.

□ Used by some models to initialize individual trust/reputation values.

□ How-to:

• mapping between the initial individual reputation of a stranger and the group from which he or she comes from.

□ Problem: highly domain-dependant and model-dependant.

Propagated:

agent can attempt to estimate the stranger's reputation based on information garnered from others in the environment. Also called *word-of-mouth*.

□ Used by several models. See [5] as example.

□ How-to: reputation values can be exchanged (recommended) from one agent to another...

a) Upon request: one agent request another agent(s) to provide their estimate (a recommendation) of the stranger's reputation, then combines the results coming from these agents depending on the recommenders' reputation

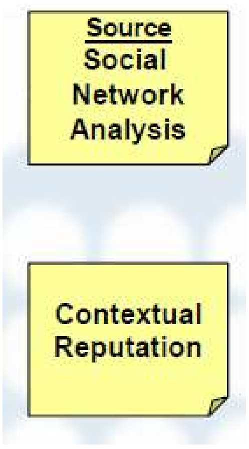
b) Propagation mechanism: some mechanism to have a distributed reputation computation.

□Problem: the combination of the different reputation values tends to be an ad-hoc solution with no social basis.

• E.g. a weighted sum of a combination of the stranger agent's reputation values and the recommender agents' reputation values

Sociological information as source

- Sabater [5] and Pujol [4] identify another source for trust/reputation: *Social relations* established between agents.
- Used only in scenarios where there is a rich interaction between agents
- How-to: usually by means of *social network analysis*
  - Detect nodes (agents) in the network that are widely used as (trusted) sources of information
  - – E.g. Google's page rank analyzes the topology of the network of links. Highly-linked pages get more reputation (nodes with high in-link ratios).
- Problem: depends on the availability of relational data



Distributed Reputation Model

- General, 'common sense' model.
- Distributed: based on recommendations.
- Very useful for multiagent systems (MAS).
- Agents exchange (recommend) reputation information about other agents.
- 'Quality' of information depends on the recommender's reputation.
- 'Loose' areas
- Trust calculation algorithm too ad hoc.
- Lacking a concrete definition of trust for distributed systems.

- **Trust Model Overview**

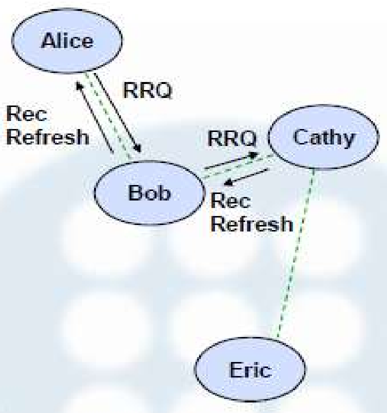
- 1-to-1 asymmetric trust relationships.
- Direct trust and recommender trust.
- Trust categories and trust values [-1,0,1,2,3,4].

Conditional transitivity.

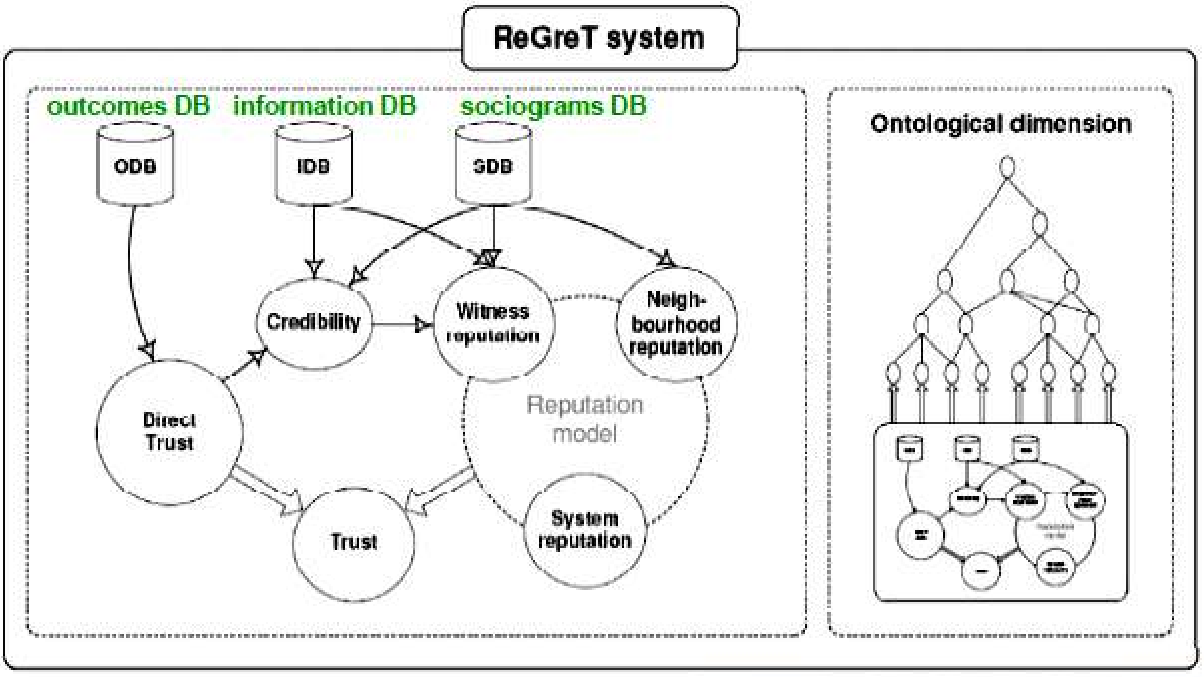
- *Alice* trusts *Bob* .&. *Bob* trusts *Cathy* ⇒ *Alice* trusts *Cathy*
- *Alice* trusts.rec *Bob* .&. *Bob* says *Bob* trusts *Cathy* ⇒ *Alice* may trust *Cathy*
- *Alice* trusts.rec *Bob* value *X* .&. *Bob* says *Bob* trusts *Cathy* value *Y*

⇒ *Alice* may trust *Cathy* value f(*X*,*Y*)

- Recommendation protocol
    1. Alice → Bob: RRQ(Eric)
    2. Bob → Cathy: RRQ(Eric)
    3. Cathy → Bob: Rec(Eric,3)
    4. Bob → Alice: Rec(Eric,3)
- Refreshing recommendations
    1. Cathy → Bob: Refresh(Eric,0)
    2. Bob → Alice: Refresh(Eric,0)



Sabater's ReGreT model

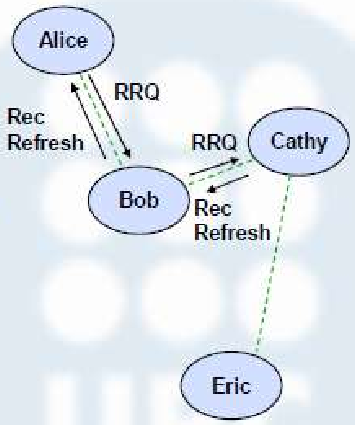


- Calculating Trust (1 path)
    - $tv_p(T) = tv(R1)/4 \times tv(R2)/4 \times .. \times tv(Rn)/4 \times rtv(T)$

      trust value (for known agents)

      recommended trust value (for stranger agents)

    - **E.g: $tv_p(Eric)$**
      $= tv(Bob)/4 \times tv(Cathy)/4 \times rtv(Eric)$
      $= 3/4 \times 2/4 \times 3$
      $= 1.12$



 The system maintains three knowledge bases:
 the *outcomes data base* (*ODB*) to store previous contracts and their result
 the *information data base* (*IDB*), that is used as a container for the information received from other partners

 the *sociograms data base* (*SDB*) to store the sociograms that define the agent social view of the world.

 These data bases feed the different modules of the system.

 In the ReGreT system, each trust and reputation value computed by the modules has an associated reliability measure

***Direct Trust*:**

 ReGreT assumes that there is no difference between *direct interaction* and *direct observation* in terms of reliability of the information. It talks about *direct experiences*.

 The basic element to calculate a *direct trust* is the *outcome*.

 An *outcome* of a dialog between two agents can be either:

- An initial contract to take a particular course of action and the actual result of the actions taken, or
- An initial contract to x the terms and conditions of a transaction and the actual values of the terms of the transaction.

 Reputation Model: *Witness reputation*

 First step to calculate a witness reputation is to *identify the set of witnesses* that will be taken into account by the agent to perform the calculation.

 The initial set of potential witnesses might be

- the set of all agents that have interacted with the target agent in the past.
- This set, however, can be very big and the information provided by its members probably suffer from the *correlated evidence problem.*

 Next step is to aggregate these values to obtain a single value for the *witness reputation*.

 The importance of each piece of information in the final reputation value will be proportional to the *witness credibility.*

 Two methods to evaluate *witness credibility*:

• ReGreT uses *fuzzy rules* to calculate how the structure of social relations influences the credibility on the information. The antecedent of each rule is the type and degree of a social relation (the edges in a sociogram) and the consequent is the credibility of the witness from the point of view of that social relation. E.g.,

• The second method used in the ReGreT system to calculate the credibility of a witness is to *evaluate the accuracy of previous pieces of information* sent by that witness to the agent. The agent is using the *direct trust* value to measure the truthfulness of the information received from witnesses.

– E.g., an agent A receives information from witness W about agent B saying agent B offers good quality products. Later on, after interacting with agent B
realizes that the products that agent B is selling are horrible

 Reputation Model: *Neighbourhood Reputation*

 Neighbourhood in a MAS is not related with the physical location of
the agents but with created interaction.

 The main idea is that the behaviour of these neighbours and the kind of relation they have with the target agent can give some clues about the behaviour of the target agent.

 To calculate a *Neighbourhood Reputation* the ReGreT system uses *fuzzy rules*.

• The antecedents of these rules are one or several *direct trusts*

associated to different behavioural aspects and the relation between the target agent and the neighbour.
- The consequent is the value for a concrete reputation (that can be associated to the same behavioural aspect of the trust values or not).

Reputation Model: ***System Reputation***

- to use the common knowledge about *social groups* and the role that the agent is playing in the society as a mechanism to assign default reputations to the agents.
- ReGreT assumes that the members of these groups have one or several *observable* features that unambiguously identify their membership.
- Each time an agent performs an action we consider that it is playing a single role.
- E.g. an agent can play the role of buyer and seller but when it is

selling a product only role seller relevant. nly the role of seller is relevant
- *System reputations* are calculated using a table for each social group where the rows are the roles the agent can play for that group, and the columns the behavioural aspects.

Reputation Model: ***Default Reputation***

- To the previous reputation types we have to add a fourth one, the reputation assigned to a third party agent when there is no information at all: the *default* reputation.
- Usually this will be a fixed value

Reputation Model: **Combining reputations**

- Each reputation type has different characteristics and there are a lot of *heuristics* that can be used to aggregate the four reputation values to obtain a single and representative reputation value.
- In ReGreT this heuristic is based on the default and calculated reliability assigned to each type.
- Assuming we have enough information to calculate all the reputation types, we have the stance that
- witness reputation is the first type that should be considered, followed by
- the *neighbourhood reputation*,
- *system reputation*
- the *default reputation*.
- This ranking, however, has to be subordinated to the calculated reliability for each type.

Uses and Drawbacks

- Most Trust and Reputation models used in MAS are devoted to
  - *Electronic Commerce*
  - *Recommender and Collaborative Systems*
  - *Peer-to-peer file-sharing systems*
- Main criticism to Trust and Reputation research:
  - Proliferation of **ad-hoc models** weakly grounded in social theory
  - **No general, cross-domain model** for reputation
  - **Lack of integration** between models
- Comparison between models unfeasible
- Researchers are trying to solve this by, e.g. the ART competition

1. **Define Agent**

Intelligent Agent is an autonomous entity that exists in an environment and acts in a rational way.

2. **What is a ROBOT?**

- A robot can also be considered an agent.
- A robot includes a variety of sensors including vision (cameras, ultrasonic transducers, infrared detectors), hearing (microphones), touch (bump sensors), as well as other types of sensors for pressure, temperature, and movement detection (accelerometers).
- Effectors include motors (to drive wheels, tracks, or limbs), and a speaker for sound vocalization. A robot can also include a number of other effectors that can manipulate the environment, such as a vacuum, a water pump, or even a weapon

3. **Identify the components of an agent**

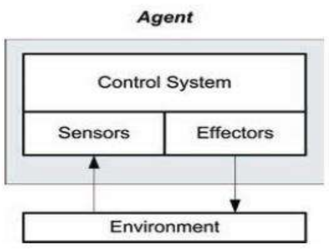


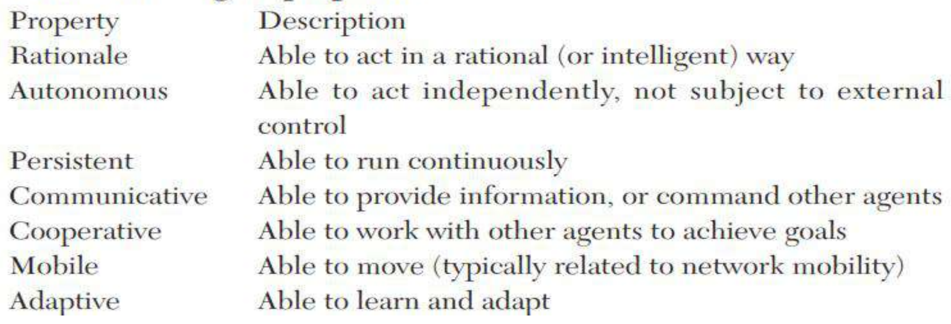
FIGURE 11.1: **The fundamental anatomy of an agent.**

An agent includes

- One or more sensors that are used to perceive the environment,
- One or more effectors that manipulate the environment,
- And a control system. The control system provides a mapping from sensors to effectors, and provides intelligent (or rational) behaviour.

4. **Classify the properties of and agent.**

Table 11.1: Agent properties.



Property	Description
Rationale	Able to act in a rational (or intelligent) way
Autonomous	Able to act independently, not subject to external control
Persistent	Able to run continuously
Communicative	Able to provide information, or command other agents
Cooperative	Able to work with other agents to achieve goals
Mobile	Able to move (typically related to network mobility)
Adaptive	Able to learn and adapt

5. **Categorize the agent environments**

Observability, Dynamic, Deterministic, Episodic, ContinuousMulti-Agent

6. Mention the data structure of agent communication protocol

A protocol is specified by a **data structure with the following five fields**:

1.sender

2.receiver(s)

3.language in the protocol

4.encoding and decoding functions

5.actions to be taken by the receiver(s)

7. Identify the 3 aspects of speech act.

1.**Locution**, the physical *utterance* by the speaker

2.**Illocution**, the *intended meaning* of the utterance by the speaker

3. **Perlocution**, the *action* that results from the locution

8. write the KQML protocol structure

(KQML-performative
:sender<word>
:receiver<word>
:language<word>
:ontology<word>
:content<expression>
...)

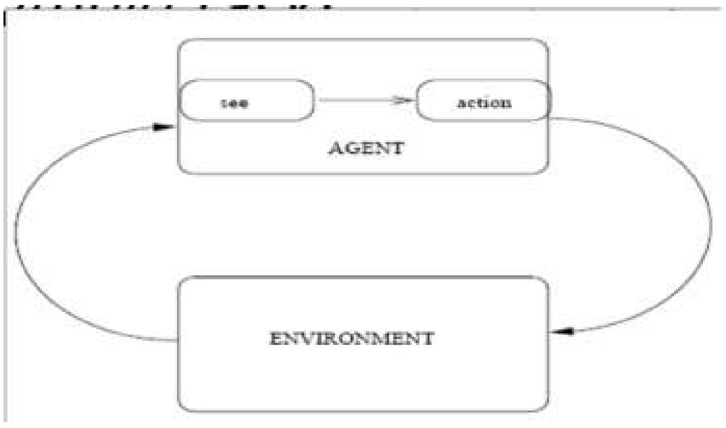
9. List out the KQML-performatives

:content(the message itself),
:language(the language in which the message is expressed),and
:ontology(the vocabulary of the "words" in the message).

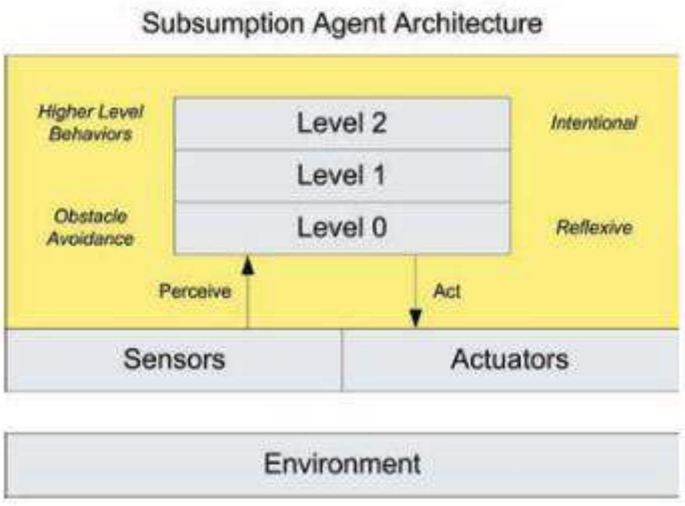
10. Define Purely Reactive Agents.

The simplest kind of agent, appropriate for simple tasks

Input: sensor data and received messages • Output: effector signals and sent messages • The most basic reactive agents are specified by a set of independent situation-action rules.

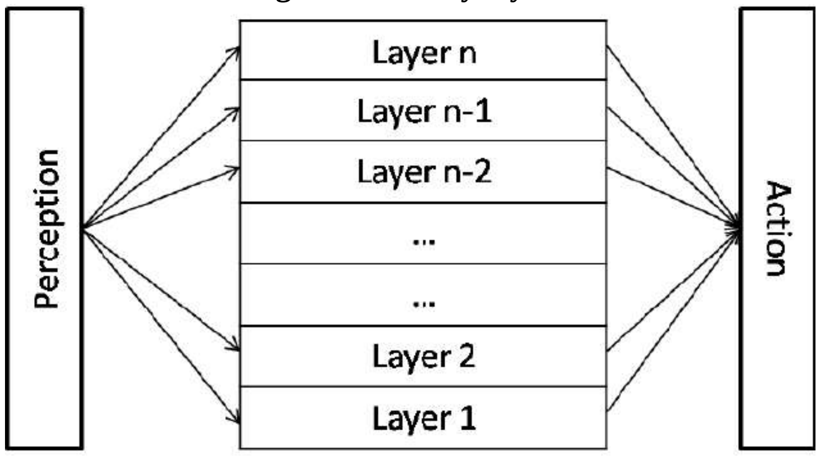


11. What are characteristics of the subsumption architecture?



Instead of guiding behavior by symbolic mental representations of the world, subsumption rchitecture couples sensory information to action selection in an intimate and bottom-up fashion. It does this by decomposing the complete behavior into sub-behaviors. These sub-behaviors are organized into a hierarchy of layers.

12. State the advantage of vertically layered architecture.



The main **advantage of vertical layered architecture** is the interaction between **layers** is reduced significantly to m 2 (n−1). The main disadvantage is that the **architecture** depends on its robustness, so if one **layer** fails, the entire system fails.

13. List the types of agents?

- Arranged in order of increasing generality:
- Simple reflex agentsv
- Model-based reflex agentsv
- Goal-based agentsv
- Utility-based agents; and
- Learning agents

14. Identify the drawbacks of table driven agents.

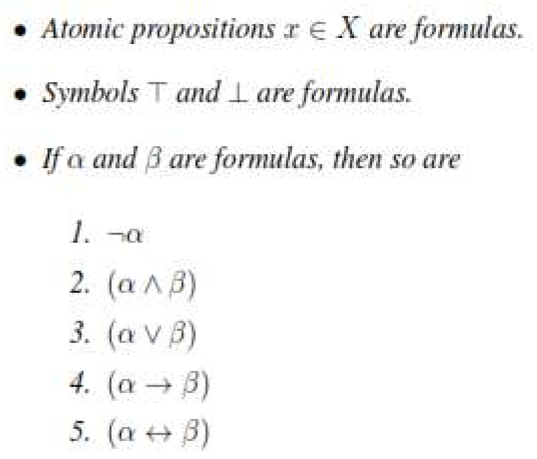
Table driven agents have

- huge table
- take a long time to construct such a tablev
- no autonomyv
- Even with learning, need a long time tov learn the table entries

15. What are logical formulae and logical deduction?

- Logical deduction is finding new facts which logically follow from the assumptions.

logical formulae:



- *Atomic propositions $x \in X$ are formulas.*
- *Symbols $\top$ and $\bot$ are formulas.*
- *If α and β are formulas, then so are*

  1. $\neg\alpha$
  2. $(\alpha \wedge \beta)$
  3. $(\alpha \vee \beta)$
  4. $(\alpha \rightarrow \beta)$
  5. $(\alpha \leftrightarrow \beta)$

16. What are the unsolved problems with other purely reactive architectures?

- No principled method exists for building such agents
- Effective agents can be generated with small numbers of behaviors, it is more difficult to build agents that

contain many layers

- The interaction between the different behaviors becomes too complex to understand

17. Define belief-desire-intention (BDI) architectures

BDI, which stands for Belief-Desire-Intention, is an architecture that follows the theory of human reasoning as defined by Michael Bratman.

Belief represents the view of the world by the agent (what it believes to be the state of the environment in which it exists).

Desires are the goals that define the motivation of the agent (what it wants to achieve). The agent may have numerous desires, which must be consistent.

Intentions specify that the agent uses the Beliefs and Desires in order to choose one or more actions in order to meet the desires

18. What are the two types of control flow within layered architectures?

A **hierarchical control system (HCS)** is a form of control system in which a set of devices and governing software is arranged in a hierarchical tree.

19. Define Agent Communication.

An agent is an active object with the ability to perceive, reason, and act. Agent has the ability to communicate. This ability is part perception (the receiving of messages) and part action (the sending of messages). Agents communicate in order to achieve better goals of themselves or of the society/system in which they exist.

20. Define Coherence.

- Coherence is how well a system behaves as a unit. A problem for a multiagent system is how it can maintain global coherence without explicit global control.
- The agents must be able on their own to determine goals they share with other agents, determine common tasks, avoid unnecessary conflicts, and pool knowledge and evidence. It is helpful if there is some form of organization among the agents.
- Social commitments can be a means to achieving coherence

21. Define the property of Coordination

- Coordination is a property of a system of agents performing some activity in a shared environment.
- The degree of coordination is the extent to which they avoid extraneous activity by reducing resource contention, avoiding livelock and deadlock, and maintaining applicable safety conditions.
- Cooperation is coordination among nonantagonistic agents, while negotiation is coordination among competitive or simply self-interested agents.
- To cooperate successfully, each agent must maintain a model of the other agents, and also develop a model of future interactions. This presupposes sociability.

22. What are the three aspects to the formal study of communication?

1. **Locution**, the physical *utterance* by the speaker
2. **Illocution**, the *intended meaning* of the utterance by the speaker
3. **Perlocution**, the *action* that results from the locution

23. Define Ontology.

In **AI, an ontology** is a specification of the meanings of the symbols in an information system. It is a specification of what individuals and relationships are assumed to exist and what terminology is used for them.

24. Define bargaining.

In a bargaining setting, **agents can make a mutually beneficial agreement**, but have a conflict of interest about which agreement to make.

There are two major subfields of bargaining theory: axiomatic and strategic.

25. Define agent architecture.

Agent architectures, like software architectures, are formally a description of the elements from which a system is built and the manner in which they communicate. Further, these elements can be defined from patterns with specific constraints.

Example :

Reactive Architectures

Deliberative Architectures

Blackboard Architectures

PART B

1. What are Abstract Architectures for Intelligent Agents.(13)
2. Write briefly on Concrete Architectures for Intelligent Agents.(13)
3. Write a short note on Layered architectures. (13)
4. Define Agent Communication. Write a short note on coordination, Dimensions of meaning and Message types.(13)
5. Explain Negotiation in detail. (13)
6. Explain Bargaining theories in detail. (13)
7. Narrate Argumentation among Agents in detail.(13)
8. Briefly explain

(i). Communication Levels (4)

(ii). Speech Acts (3)

(iii). Knowledge Query and Manipulation Language (KQML)(3)

(iv). Knowledge Interchange Format (KIF)(3)

9. With diagrammatic representation, explain Trust and Reputation in Multi-agent systems in detail.(13)
10. Compare and contrast about the negotiation and bargaining.(13)
11. Examine the Argumentation among Agents.(13)
12. Describe the trust and reputation in multi-agent systems.(13)

PRATHYUSHA ENGINEERING COLLEGE

Department of Computer Science and Engineering

VI Semester (EVEN 2019-20)

CS8691ARTIFICIAL INTELLIGENCE

UNIT V APPLICATIONS 9

AI applications –Language Models –Information Retrieval-Information Extraction –Natural Language Processing -Machine Translation –Speech Recognition –Robot –Hardware –Perception –Planning –Moving

1. AI APPLICATIONS

AI programs are developed to perform specific tasks, that is being utilized for a wide range of activities including *medical diagnosis, electronic trading platforms, robot control, and remote sensing*. AI has been used to develop and advance *numerous fields and industries, including finance, healthcare, education, transportation, and more*.

- Information Retrieval
- Natural Language Processing
- Machine Translation
- Speech Recognition
- Robot Marketing
- Banking
- Finance
- Agriculture
- HealthCare
- Gaming
- Space Exploration
- Autonomous Vehicles
- Chatbots
- Artificial Creativity

Marketing

- we search for an item on any e-commerce store, we get all possible results related to the item. It's like these search engines read our minds! In a matter of seconds, we get a list of all relevant items.

With the growing advancement in AI, in the near future, it may be possible for consumers on the web to buy products by snapping a photo of it

Banking

- A lot of banks have already adopted AI-based systems to provide customer support, detect anomalies and credit card frauds. An example of this is HDFC Bank.
- HDFC Bank has developed an AI-based chatbot called *EVA* (Electronic Virtual Assistant), built by Bengaluru-based Senseforth AI Research.
- Since its launch, Eva has addressed over 3 million customer queries, interacted with over half a million unique users, and held over a million conversations.
- Eva can collect knowledge from thousands of sources and provide simple answers in less than 0.4 seconds.
- AI solutions can be used to enhance security across a number of business sectors, including retail and finance.
- By tracing card usage and endpoint access, security specialists are more effectively preventing fraud. Organizations rely on AI to trace those steps by analyzing the behaviors of transactions.

Companies such as MasterCard and RBS WorldPay have relied on AI and *Deep Learning* to detect fraudulent transaction patterns and prevent card fraud for years now. This has saved millions of dollars.

Finance

- Ventures have been relying on computers and data scientists to determine future patterns in the market. Trading mainly depends on the ability to predict the future accurately.
- Machines are great at this because they can crunch a huge amount of data in a short span. Machines can also learn to observe patterns in past data and predict how these patterns might repeat in the future.
- Financial organizations are turning to AI to improve their stock trading performance and boost profit.

Agriculture

- AI can help farmers get more from the land while using resources more sustainably.
- Issues such as climate change, population growth, and food security concerns have pushed the industry into seeking more innovative approaches to improve crop yield.
- Organizations are using automation and robotics to help farmers find more efficient ways to protect their crops from weeds.
- Blue River Technology has developed a robot called See & Spray which uses computer vision technologies like *object detection* to monitor and precisely spray weedicide on cotton plants. Precision spraying can help prevent herbicide resistance
- The image recognition app identifies possible defects through images captured by the user's smartphone camera. Users are then provided with soil restoration techniques, tips, and other possible solutions. The company claims that its software can achieve pattern detection with an estimated accuracy of up to 95%.

Health Care

- When it comes to saving our lives, a lot of organizations and medical care centers are relying on AI.
- clinical decision support system for stroke prevention that can give the physician a warning when there's a patient at risk of having a heart stroke.
- Preventive care
- Personalized medicine

Gaming

- Artificial Intelligence has become an integral part of the gaming industry. In fact, one of the biggest accomplishments of AI is in the gaming industry.
- DeepMind's AI-based AlphaGo software, which is known for defeating Lee Sedol, the world champion in the game of GO, is considered to be one of the most significant accomplishment in the field of AI
- The actions taken by the opponent AI are unpredictable because the game is designed in such a way that the opponents are trained throughout the game and never repeat the same mistakes.
- They get better as the game gets harder. This makes the game very challenging and prompts the players to constantly switch strategies and never sit in the same position.

Space Exploration

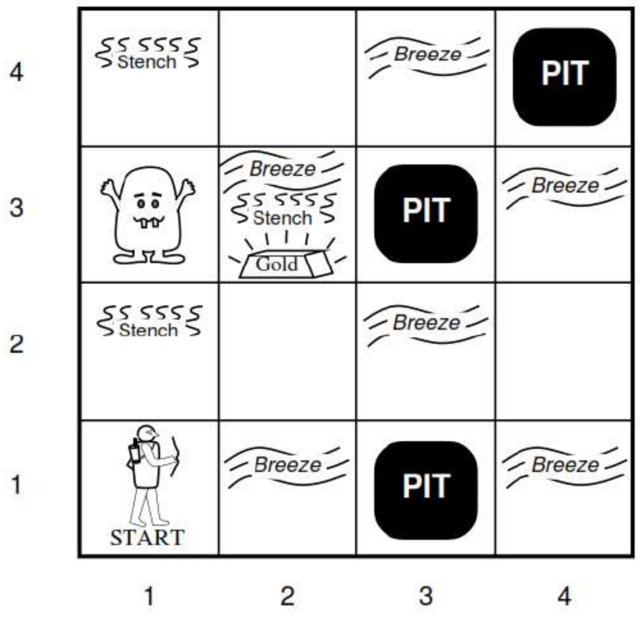
- Space expeditions and discoveries always require analyzing vast amounts of data.
- Artificial Intelligence and Machine learning is the best way to handle and process data on this scale.
- After rigorous research, astronomers used Artificial Intelligence to sift through years of data obtained by the Kepler telescope in order to identify a distant eight-planet solar system.

- Artificial Intelligence is also being used for NASA's next rover mission to Mars, the Mars 2020 Rover. The AEGIS, which is an AI-based Mars rover is already on the red planet.
- The rover is responsible for autonomous targeting of cameras in order to perform investigations on Mars.

Autonomous Vehicles

- For the longest time, self-driving cars have been a buzzword in the AI industry. The development of autonomous vehicles will definitely revolutionaries the transport system.
- Companies like Waymo conducted several test drives in Phoenix before deploying their first AI-based public ride-hailing service.
- The AI system collects data from the vehicles radar, cameras, GPS, and cloud services to produce control signals that operate the vehicle.
- Advanced Deep Learning algorithms can accurately predict what objects in the vehicle's vicinity are likely to do. This makes Waymo cars more effective and safer.
- Another famous example of an autonomous vehicle is Tesla's self-driving car. Artificial Intelligence implements computer vision, image detection and deep learning to build cars that can automatically detect objects and drive around without human intervention

2. WUMPUS WORLD



- The Wumpus world is a simple world example to illustrate the worth of a knowledge-based agent and to represent knowledge representation. It was inspired by a video game Hunt the Wumpus by Gregory Yob in 1973.
- The Wumpus world is a cave which has **4/4 rooms connected with passageways**. So there are total 16 rooms which are connected with each other.
- We have a knowledge-based agent who will go forward in this world. The cave has a room with a *beast which is called Wumpus, who eats anyone who enters the room*.
- The Wumpus can be shot by the agent, but the agent has a single arrow.
- In the Wumpus world, *there are some Pits rooms which are bottomless,* and if agent falls in Pits, then he will be stuck there forever.
- The exciting thing with this cave is that in one room there is a possibility of finding a heap of gold. So the agent goal is *to find the gold and climb out the cave* without fallen into Pits or eaten by Wumpus.
- The agent will get a reward if he comes out with gold, and he will get a penalty if eaten by Wumpus or falls in the pit.

There are also some components which can help the agent to navigate the cave. These components are given as follows:

a. The rooms *adjacent to the Wumpus room are smelly*, so that it would have some stench.
b. The *room adjacent to PITs has a breeze,* so if the agent reaches near to PIT, then he will perceive the breeze.
c. There will be *glitter in the room if and only if the room has gold*.
d. The Wumpus can be killed by the agent if the agent is facing to it, and Wumpus will emit a horrible scream which can be heard anywhere in the cave.

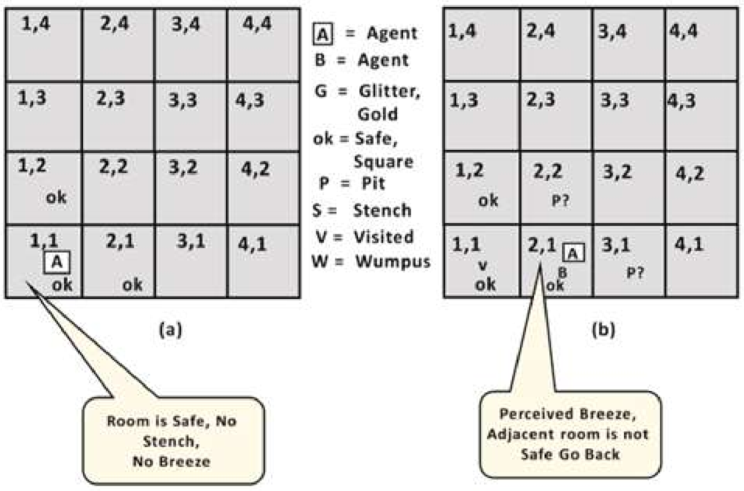
The Wumpus world Properties:

- **Partially observable:** The Wumpus world is partially observable because the agent can only perceive the close environment such as an adjacent room.
- **Deterministic:** It is deterministic, as the result and outcome of the world are already known.
- **Sequential:** The order is important, so it is sequential.
- **Static:** It is static as Wumpus and Pits are not moving.
- **Discrete:** The environment is discrete.
- **One agent:** The environment is a single agent as we have one agent only and Wumpus is not considered as an agent.

Exploring the Wumpus world:

- Now we will explore the Wumpus world and will determine how the agent will find its goal by applying logical reasoning.

  Agent's First step:
- Initially, the agent is in the first room or on the square [1,1], and we already know that this room is safe for the agent, so to represent on the below diagram (a) that room is safe we will add symbol OK. Symbol A is used to represent agent, symbol B for the breeze, G for Glitter or gold, V for the visited room, P for pits, W for Wumpus.
- At Room [1,1] agent does not feel any breeze or any Stench which means the adjacent squares are also OK.

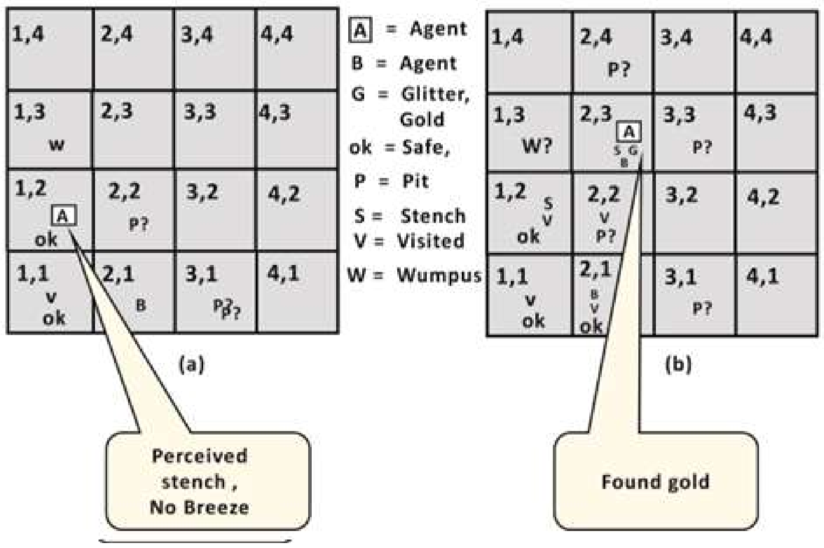


Agent's second Step:
- Now agent needs to move forward, so it will either move to [1, 2], or [2,1]. Let's suppose agent moves to the room [2, 1], at this room agent perceives *some breeze which means Pit is around this room*. The pit can be in [3, 1], or [2,2], so we will add symbol P? to say that, is this Pit room?

- Now agent will stop and think and will not make any harmful move. The agent will go back to the [1, 1] room. The room [1,1], and [2,1] are visited by the agent, so we will use symbol V to represent the visited squares.

Agent's third step:

- At the third step, now agent will move to the room [1,2] which is OK. In the room [1,2] agent perceives a stench which means there must be a Wumpus nearby. But Wumpus cannot be in the room [1,1] as by rules of the game, and also not in [2,2] (Agent had not detected any stench when he was at [2,1]). Therefore agent infers *that Wumpus is in the room [1,3], and in current state*, there is no breeze which means in [2,2] there is no Pit and no Wumpus. So it is safe, and we will mark it OK, and the agent moves further in [2,2].



Agent's fourth step:

- At room [2,2], here no stench and no breezes present so let's suppose agent decides to move to [2,3]. At room [2,3] agent perceives glitter, *so it should grab the gold and climb out of the cave*.

3. LANGUAGE MODELS

Humans use a limited number of conventional signs (smiling, shaking hands) to communicatein much the same way as other animals. Humans have also developed a complex, structuredsystem of signs known as **language** that enables them to communicate most of what they knowabout the world.

One of the actions that is available to an agent is to produce language.
This is called a **speechact.** "Speech" is used in the same sense as in "free speech," not "talking," so typing, skywriting,and using sign language all count as speech acts.
English has no neutral word for an agent thatproduces language, either by speaking or writing or anything else. We will use **speaker, hearer, a**nd**utterance** as generic terms referring to any mode of communication.

Imagine a group of **agents are exploring the wumpus world together**. The group gains anadvantage (collectively and individually) by being able to do the following:

- **Inform** each other about the part of the world each has explored, so that each agent hasless exploring to do. This is done by making statements: *There's a breeze here in 3 4.*
- **Query** other agents about particular aspects of the world. This is typically done by askingquestions: *Have you smelled the wumpus anywhere?*
- **Answer** questions. This is a kind of informing. *Yes, I smelled the wumpus in 2 5.*
- **Request or command** other agents to perform actions: *Please help me carry the gold.* Itcan be seen as impolite to make a direct requests, so often an **indirect speech act** (a requestin the form of a

statement or question) is used instead: / *could use some help carrying this*or*Could you help me cany this?*

- **Promise** to do things or **offer** deals: /'// *shoot the wumpus if you let me share the gold.*
- **Acknowledge** requests and offers: *OK.*
- **Share** feelings and experiences with each other: *You know, old chap, when I get in a spotlike that, I usually go back to the start and head out in another direction,* or *Man, thatwumpus sure needs some deodorant!*

Planning problem

The hard part for an agent is to decide *when* a speech act of some kind is called for,and to decide *which* speech act, out of all the possibilities, is the right one. At one level,this is just the familiar **planning** problem—an agent has a set of possible actions to choosefrom, and must somehow try to choose actions that achieve the goal of communicating someinformation to another agent.

Understanding problems

- The problem of understanding speech acts is much like other **understanding** problems,such as *image understanding or medical diagnosis*.
- Part of the speech act *understanding problem is specific to language*.
- We need to know something about the *syntax and semantics of a language* to determine why anotheragent performed a given speech act.
- The understanding problem also includes the more generalproblem of **plan recognition.**
- Part of the understanding problem can be *handled by logical reasoning*.
- We will see thatlogical implications are a good way of describing the ways that words and phrases combine toform larger phrases. Another part of the understanding problem can only be handled by uncertainreasoning techniques.
- Usually, there will be several states of the world that could all lead to thesame speech act, so the understander has to decide which one is more probable.

Fundamentals of language

Formal languages—the ones like Lisp and first-order logic that areinvented and rigidly defined

Natural languages—the ones like Chinese, Danish, andEnglish that humans use to talk to one another.

Formal language

- A formal language is defined as a set of strings, where each string is a sequence of symbolstaken from a finite set called the **terminal symbols.**
- most of them are similar in that they are based onthe idea of **phrase structure**
- Strings are composed of substrings called **phrases,** whichcome in different categories.
- Categorizing phrases helps us to describe the allowable strings of thelanguage.
- Any of the noun phrases can combine with a **verb phrase** (or *VP)* suchas "is dead" to form a phrase of category **sentence**
- **Grammatical categories** are essentially posited as part of ascientific theory of language that attempts to account for the difference between grammatical andungrammatical categories.

Example:

noun phrase"the wumpus," "the king,"
verb phrase : "is dead"
sentence :"the wumpus is dead"
Grammatical categories :"the wumpus is dead" , "wumpus the dead is"

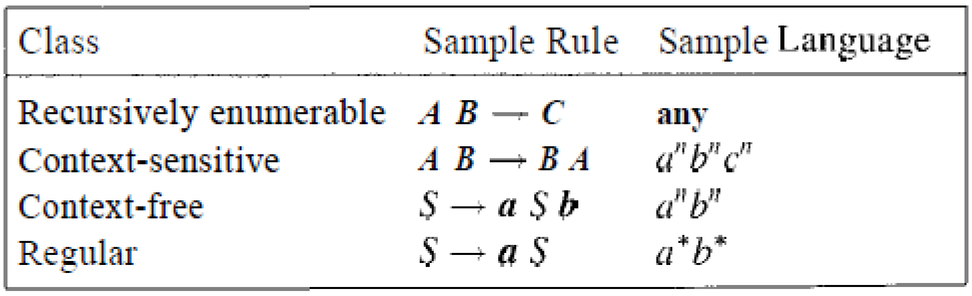
Categories such as *NP, VP,* and *S* are called **nonterminal symbols.** In the BNF notation,**rewrite rules** consist of a single nonterminal symbol on the left-hand side, and a sequence ofterminals or nonterminals on the right-hand side

The meaning of a rule such as S— *NP VP*

Grammatical formalisms can be classified by their **generative capacity:** the set oflanguages they can representfour classes of grammaticalformalisms that differ only in the form of the rewrite rules.

- **Recursively enumerable** grammars use unrestricted rules: both sides of therewrite rules can have any number of terminal and nonterminal symbols. Thesegrammars are equivalent to Turing machines in their expressive power.
- **Context-sensitive grammars** are restricted only in that the right-hand hand sidemust contain at least as many symbols as the left-hand side. The name contextsensitivecomes from the fact that a rule such as
  ASB — AXB
  says that an *S* can berewritten as an *X* in the context of a preceding *A* and a following *B.*
- **In context-free grammars** (or CFGs), the left-hand side consists of a singlenonterminal symbol. Thus, ***each rule licenses rewriting the nonterminal as the righthandside in any context***. CFGs are popular for natural language grammars, althoughit is now widely accepted that at least some natural languages are not context-free.
- **Regular** grammars are the most restricted class. Every rule has a single nonterminalon the left-hand side, and a terminal symbol optionally followed by a nonterminalon the right-hand side. *Regular grammars are equivalent in power to finite-statemachines*. They are poorly suited for programming languages because, for example,they cannot represent constructs such as balanced opening and closing parentheses.

To give you an idea of which languages can be handled by which classes, thelanguage*a"b"* (a sequence of *n* copies of *a* followed by the same number of *b)* canbe generated by a context-free grammar, but not a regular grammar. The language*a"b"c"* requires a context-sensitive grammar, whereas the language *a*b** (a sequenceof any number of *a* followed by any number of *b)* can be described by any of the fourclasses. A summary of the four classes follows.



Class	Sample Rule	Sample Language
Recursively enumerable	$A\ B \rightarrow C$	any
Context-sensitive	$A\ B \rightarrow B\ A$	$a^n b^n c^n$
Context-free	$S \rightarrow a\ S\ b$	$a^n b^n$
Regular	$S \rightarrow a\ S$	$a^* b^*$

The component steps of communication

A typical communication episode, in which speaker *S* wants to convey proposition *P* to hearer *H* using words *W*, is composed of seven processes.
Three take place in the speaker:
Intention: *S* wants *H* to believe *P* (where *S* typically believes *P)*
Generation: *S* chooses the words *W* (because they express the meaning *P)*
Synthesis: 5 utters the words *W* (usually addressing them to *H)*

Four take place in the hearer:
Perception: *H* perceives *W* (ideally *W'* = *W*, but misperception is possible)
Analysis: *H* infers that *W* has possible meanings *P\,..., P,,* (words andphrases can have several meanings)
Disambiguation: *H* infers that *S* intended to convey P, (where ideally *Pj* = *P,*but misinterpretation is possible)
Incorporation: *H* decides to believe P, (or rejects it if it is out of line with
what*H* already believes)

Intention :

- Speaker decides that there is something that is worth saying tothe hearer.
- This often involves *reasoning about the beliefs and goals of the hearer*, so that theutterance will have the desired effect.
- For our example, the speaker has the intention of havingthe hearer know that the wumpus is no longer alive.

Generation :

- The speaker uses knowledge about language to decide what to say.
- This is harder than the inverse problem of understanding (i.e., analysis and disambiguation).Generation has not been stressed as much as understanding in AI, mainly because we humansare anxious to talk to machines, but are not as excited about them talking back.
- For now, we justassume the hearer is able to choose the words "The wumpus is dead."

Synthesis :

- Most language-based AI systems synthesize typed output on a screen or paper,which is a trivial task.
- Speech synthesis has been growing in popularity, and some systems arebeginning to sound human.
- The agent in the below example is synthesizing a string of soundswritten in the phonetic alphabet "[thaxwahmpahsihzdeyd]."
- The sounds that get synthesized are *different fromthe words that the agent generates*.
- Also note that the words are run together; this is typical ofquickly spoken speech.

Perception:

When the medium is speech, the perception step is called **speech recognition;**when it is printing, it is called **optical character recognition.** Both have moved from being

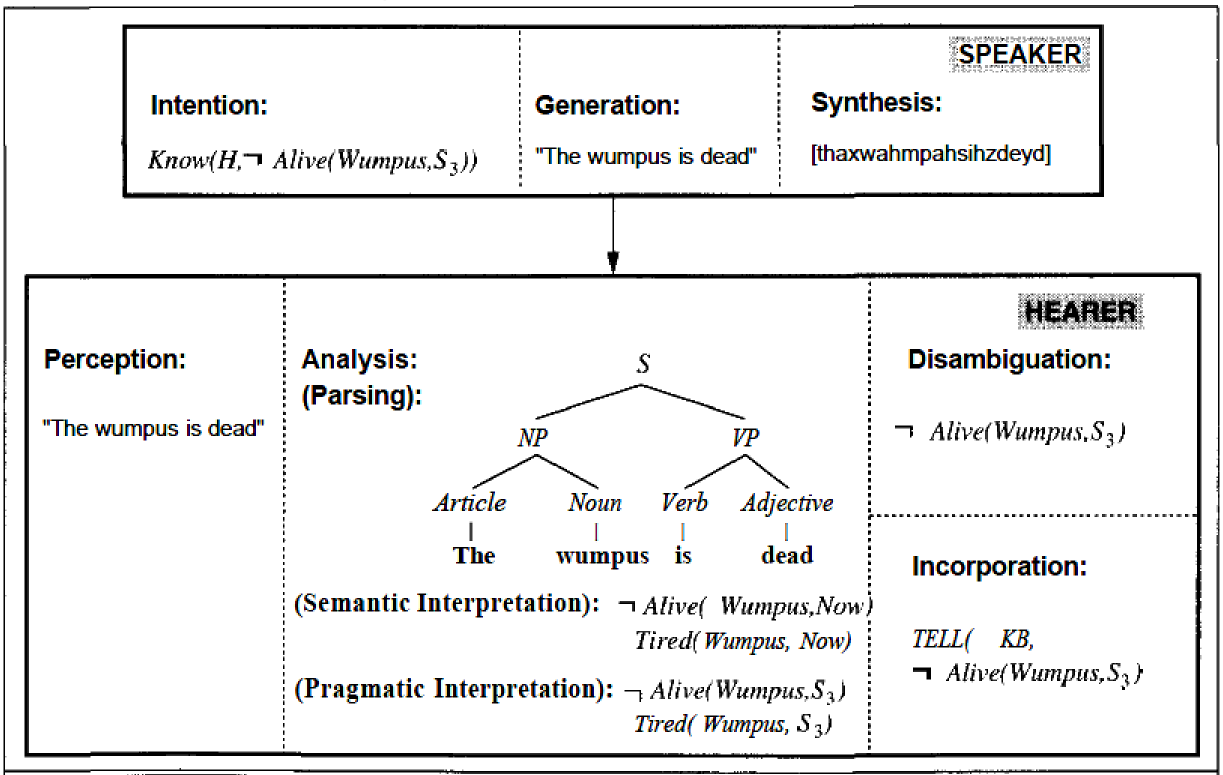


Figure 22.1 Seven processes involved in communication, using the example sentence "The wumpus is dead."

esoteric to being commonplace within the last five years. For the example, let us assume that the hearer perceives the sounds and recovers the spoken words perfectly.

Analysis:
Two main parts:
syntactic interpretation (or parsing) and semantic interpretation.

- **Semantic interpretation** includes both understanding the meanings of words and incorporating knowledge of the current situation (also called pragmatic interpretation).
- The word **parsing** is derived from the the Latin phrase *pars orationis,* or "part of speech," and refers to the process of assigning a part of speech (noun, verb, and so on) to each word in a sentence and grouping the words into phrases.
- One way of displaying the result of a syntactic analysis is with a **parse tree.**
- *A parse tree is a tree in which interior nodes represents phrases, links represent applications of grammar rules, and leaf nodes represent words*.
- We define **the yield of a node as the list of all the leaves below the node**, in left-to-right order, then we can say that the meaning of a parse tree is that each node with label *X* asserts that the yield of the node is a phrase of category *X.*
- **Semantic interpretation is the process of extracting the meaning of an utterance as an expression in some representation language.**
- Two possible semantic interpretations: that the wumpus is not alive, and that it is tired (a colloquial meaning of *dead).*

Ambiguity

- Utterances with several possible interpretations are said to be **ambiguous.**
- **lexical ambiguity,** where a word has more than one meaning.
- **Syntactic ambiguity** (also known as **structural ambiguity)** can occur with or without lexical ambiguity.

For example, the string "I smelled a wumpus in 2,2" has two parses: one where the propositional phrase modifies the noun, and one where it modifies the verb.

- **semantic ambiguity** :The syntactic ambiguity leads to a **semantic ambiguity,** because one parse means the wumpus is in 2,2 and the other means that a stench is in 2,2.
- **Referential ambiguity** occurs because natural languages consist almost entirely of words for categories, not for individual objects. There is no word for *the-apple-I-had-for-lunch-today,]\\si*categories like *apple.*
- **Pragmatic ambiguity** occurs when the speaker and hearer disagree on what the current situation is. If the speaker says "I'll meet you next Friday" thinking that they're talking about the 17th, and the hearer thinks that they are talking about the 24th, then there is miscommunication.
- **Local ambiguity** :Sometimes a phrase or sentence has **local ambiguity,** where a substring can be parsed several ways, but only one of those ways fits into the larger context of the whole string. For example, in the C programming language, the string *c means "pointer to c" when it appears in the declaration char *c; but it means "multiply by c" when it appears in the expression 2 *c.

.

Disambiguation.

- Most speakers are not intentionally ambiguous, but most utterances have several legal interpretations. Communication works because the hearer does the work of figuring out which interpretation is the one the speaker probably meant to convey.
- Disambiguation is the first process that depends heavily on uncertain reasoning. Analysis generates possible interpretations; if more than one interpretation is found, then disambiguation chooses the one that is best.

1. The world model: the probability that a fact occurs in the world.
2. The mental model: the probability that the speaker forms the intention of communicating this fact to the hearer, given that it occurs.9 (This combines models of what the speaker believes, what the speaker believes the hearer believes, and so on.)
3. The language model: the probability that a certain string of words will be chosen, given that the speaker has the intention of communicating a certain fact.

4. The acoustic model: the probability that a particular sequence of sounds will be generated, given that the speaker has chosen a given string of words

Incorporation.A totally naive agent might believe everything it hears, but a sophisticated agent treats the words *W* and the derived interpretation P, as additional pieces of evidence that get considered along with all other evidence for and against P,.

It only makes sense to use language when there are agents to communicate with who
(a) understand a common language,
(b) have a shared context on which to base the conversation, and
(c) are at least somewhat rational.

TWO MODELS OF COMMUNICATION

Study of communication centers on the way that an agent's beliefs are turned into words and back into beliefs in another agent's knowledge base (or head). There are two ways of looking at ENCODED MESSAGE the process.

ENCODED MESSAGE

The **encoded message** model says that the speaker has

- a definite proposition *P* in mind, and
- encodes the proposition into the words (or signs) *W*.

The hearer then tries to

- decode the message *W* to retrieve the original proposition *P* (cf. Morse code).

Under this model, the meaning in the speaker's head, the message that gets transmitted, and the interpretation that the hearer arrives at all ideally carry the same content. When they differ, it is because of noise in the communication channel or an error in encoding or decoding.

SITUATED LANGUAGE

Limitations of the encoded message model led to the **situated language** model, which says that the meaning of a message depends on both the words and the **situation** in which the words are uttered. In this model, just as in situation calculus, the encoding and decoding functions take an extra argument representing the current situation.

This accounts for the fact that the same words can have very different meanings in different situations.

"I am here now" represents one fact when spoken by Peter in Boston on Monday, and quite another fact when spoken by Stuart in Berkeley on Tuesday.

More subtly, "You must read this book" is a suggestion when written by a critic in the newspaper, and an assignment when spoken by an instructor to the class. "Diamond" means one thing when the subject is jewelry, and another when the subject is baseball.

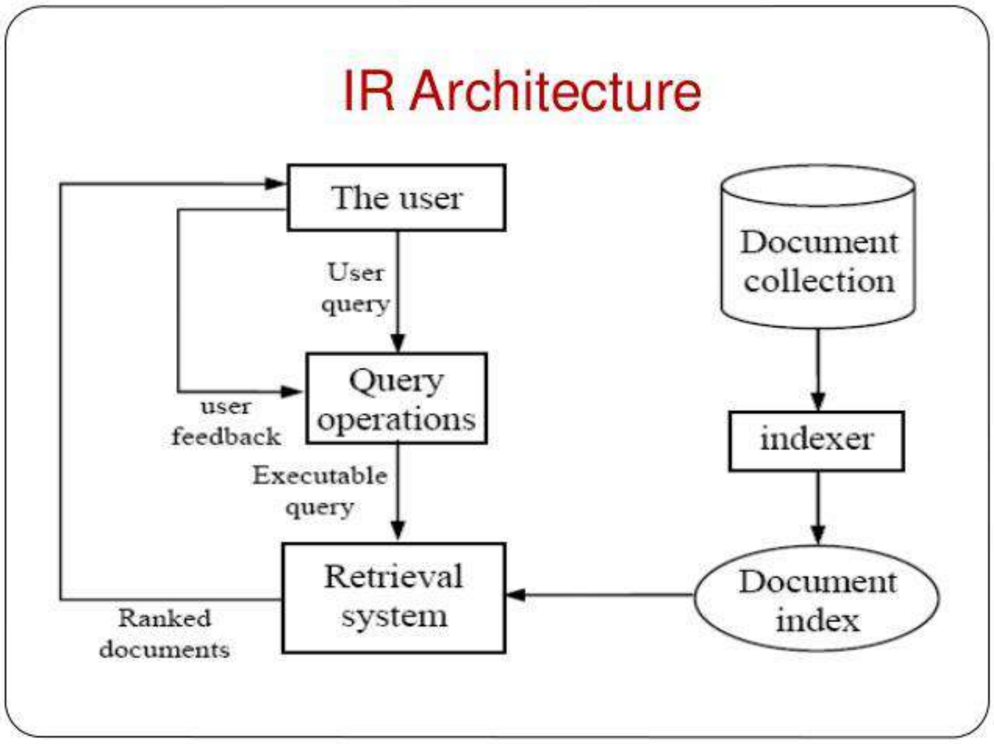
The situated language model points out a possible source of communication failure: if the speaker and hearer have different ideas of what the current situation is, then the message may not get through as intended.

For example, suppose agents *X*, *Y*, and Z are exploring the wumpus world together. *X* and *Y* secretly meet and agree that when they smell the wumpus they will both shoot it, and if they see the gold, they will grab it and run, and try to keep Z from sharing it. Now suppose*X* smells the wumpus, while *Y* in the adjacent square smells nothing, but sees the gold.

X yells "Now!," intending it to mean that now is the time to shoot the wumpus, but *Y* interprets itas meaning now is the time to grab the gold and run.

4. INFORMATION RETRIEVAL

Information retrieval is the task of finding documents that are relevant to a user's need for information. The best-known examples of information retrieval systems are search engines on the World Wide Web. A Web user can type a query such as [AI book] into a search engine and see a list of relevant pages.



An information retrieval (henceforth **IR**) system can be characterized by

1. **A corpus of documents.** Each system must decide what it wants to treat as a document: a paragraph, a page, or a multipage text.

2. **Queries posed in a query language**. A query specifies what the user wants to know.
The query language can be just a list of words, such as [AI book]; or it can specify a phrase of words that must be adjacent, as in ["AI book"]; it can contain Boolean operators as in [AI AND book]; it can include non-Boolean operators such as [AI NEAR book] or [AI book site:www.aaai.org].

3. A result set. This is the subset of documents that the IR system judges to be relevant to the query. By relevant, we mean likely to be of use to the person who posed the query, for the particular information need expressed in the query.

4. A presentation of the result set. This can be as simple as a ranked list of document titles or as complex as a rotating color map of the result set projected onto a three- dimensional space, rendered as a two-dimensional display.

Boolean keyword model

Each word in the document collection is treated as a Boolean feature that is true of a document if the word occurs in the document and false if it does not. The query language is the language of Boolean expressions over features. A document is relevant only if the expression evaluates to true.

Advantage

1. Simple to explain and implement.

Disadvantages

1. The degree of relevance of a document is a single bit, so there is no guidance as to how to order the relevant documents for presentation.
2. Boolean expressions are unfamiliar to users who are not programmers or logicians.

<u>IR scoring functions</u>

BM25 scoring function

A scoring function takes a document and a query and returns a numeric score; the most relevant documents have the highest scores. In the BM25 function, the score is a linear weighted combination of scores for each of the words that make up the query.
Three factors affect the weight of a query term:

1. **Frequency** with which a query term appears in a document (also known as TF for term frequency). For the query [farming in Kansas], documents that mention “farming” frequently will have higher scores.
2. **Inverse document frequency** of the term, or IDF . The word “in” appears in almost every document, so it has a high document frequency, and thus a low inverse document frequency, and thus it is not as important to the query as “farming” or “Kansas.”
3. **Length of the document**. A million-word document will probably mention all the query words, but may not actually be about the query. A short document that mentions all the words is a much better candidate

BM25systems create an **index** ahead of time that lists, for each vocabulary word, the documents that contain the word. This is called the **hit list** for the word. Then when given a query, we intersect the hit lists of the query words and only score the documents in the intersection.

IR system evaluation

Two measures used in the scoring: recall and precision.
Precision measures the proportion of documents in the result set that are actually relevant.
Recall measures the proportion of all the relevant documents in the collection that are in the result set
A summary of both measures is the **F_1 score**, a single number that is **the harmonic mean of precision and recall**, $2PR/(P+R)$.

IR refinements

There are many possible refinements to the system described here, and indeed Web search engines are continually updating their algorithms as they discover new approaches and as the Web grows and changes.

pivoted document length normalization scheme; the idea is that the pivot is the document length at which the old-style normalization is correct; documents shorter than that get a boost and longer ones get a penalty.

Most IR systems do **case folding** of “COUCH” to “couch,” and some use a **stemming** algorithm to reduce. “couches” to the stem form “couch,” both in the query and the documents. This typically yields a small increase in recall but can harm precision

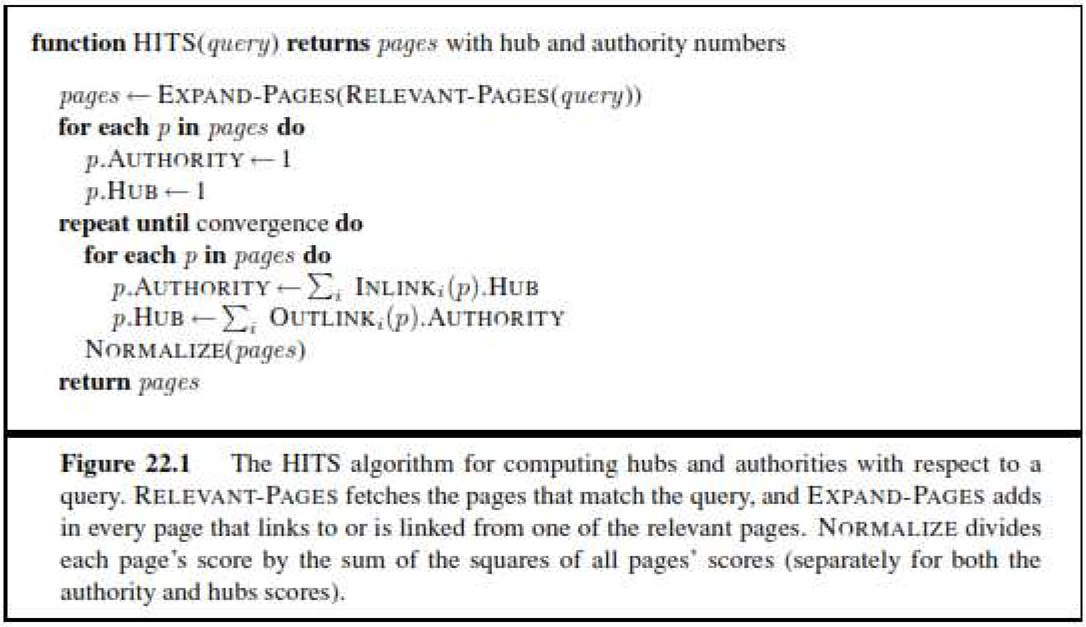
Recognize **synonyms**, such as “sofa” for “couch.”
anytime there are two words that mean the same thing, speakers of the language conspire to evolve the meanings to remove the confusion.
Related words that are not synonyms also play an important role in ranking—terms like “leather”, “wooden,” or “modern” can serve to confirm that the document really is about “couch.”

Synonyms and related words can be found in dictionaries or by looking for correlations in documents or in queries—if we find that many users who ask the query [new sofa] follow it up with the query [new couch], we can in the future alter [new sofa] to be [new sofa OR new couch]

As a final refinement, IR can be improved by considering **metadata**—data outside of the text of the document. Examples include human-supplied keywords and publication data. On the Web, hypertext **links** between documents are a crucial source of information

The PageRank algorithm

PageRank[3] was one of the two original ideas that set Google's search apart from other Web search engines when it was introduced in 1997.



```
function HITS(query) returns pages with hub and authority numbers
  pages ← EXPAND-PAGES(RELEVANT-PAGES(query))
  for each p in pages do
    p.AUTHORITY ← 1
    p.HUB ← 1
  repeat until convergence do
    for each p in pages do
      p.AUTHORITY ← $\sum_i$ INLINK$_i$(p).HUB
      p.HUB ← $\sum_i$ OUTLINK$_i$(p).AUTHORITY
    NORMALIZE(pages)
  return pages
```

Figure 22.1 The HITS algorithm for computing hubs and authorities with respect to a query. RELEVANT-PAGES fetches the pages that match the query, and EXPAND-PAGES adds in every page that links to or is linked from one of the relevant pages. NORMALIZE divides each page's score by the sum of the squares of all pages' scores (separately for both the authority and hubs scores).

PageRank was invented to solve the problem of the tyranny of TF scores: if the query is [IBM], how do we make sure that IBM's home page, `ibm.com`, is the first result, even if another page mentions the term "IBM" more frequently? The idea is that `ibm.com` has many in-links (links to the page), so it should be ranked higher: each in-link is a vote for the quality of the linked-to page.

The HITS algorithm

The Hyperlink-Induced Topic Search algorithm, also known as "Hubs and Authorities" or HITS, is another influential link-analysis algorithm.

HITS differs from PageRank in several ways.
First, it is a query-dependent measure: it rates pages with respect to a query. That means that it must be computed anew for each query—a computational burden that most search engines have elected not to take on.

Given a query, HITS first finds a set of pages that are relevant to the query.
It does that by intersecting hit lists of query words, and then adding pages in the link neighborhood of these pages—pages that link to or are linked from one of the pages in the original relevant set.

Each page in this set is considered an **authority** on the query to the degree that other pages in the relevant set point to it. A page is considered a **hub** to the degree that it points to other authoritative pages in the relevant set. Just as with PageRank, we don't want to merely count the number of links; we want to give more value to the high-quality hubs and authorities

Question answering

Information retrieval is the task of finding documents that are relevant to a query, where the query may be a question, or just a topic area or concept. **Question answering** is a somewhat different task, in which the query really is a question, and the answer is not a ranked list of documents but rather a short response—a sentence, or even just a phrase.

Once the n-grams are scored, they are filtered by expected type.

If the original query starts with
"who," then we filter on names of people;
for "how many" we filter on numbers,

for “when,” on a date or time.
There is also a filter that says the answer should not be part of the question;

5. INFORMATION EXTRACTION

Information extraction is the process of acquiring knowledge by skimming a text and looking for occurrences of a particular class of object and for relationships among objects. A typical task is to extract instances of addresses from Web pages, with database fields for street, city, state, and zip code; or instances of storms from weather reports, with fields for temperature, wind speed, and precipitation.

Six different approaches to information extraction

1. Finite-state automata for information extraction
2. Probabilistic models for information extraction
3. Conditional random fields for information extraction
4. Ontology extraction from large corpora
5. Automated template construction
6. Machine reading

Finite-state automata for information extraction

The simplest type of information extraction system is an **attribute-based extraction** system that assumes that the entire text refers to a single object and the task is to extract attributes of that object.

The problem of extracting from the text “IBM ThinkBook 970. Our price: $399.00” the set of attributes {Manufacturer=IBM, Model=ThinkBook970, Price=$399.00}. We can address this problem by defining a **tem- plate** (also known as a pattern) for each attribute we would like to extract

Templates are often defined with three parts:

1. A prefix regex,
2. A target regex, and
3. A postfix regex.

For prices, the target regex is as above, the prefix would look for strings such as “price:” and the postfix could be empty. The idea is that some clues about an attribute come from the attribute value itself and some come from the surrounding text.

If a regular expression for an attribute matches the text exactly once, then we can pull
out the portion of the text that is the value of the attribute. If there is no match, all we can do is give a default value or leave the attribute missing; but if there are several matches, we need a process to choose among them. One strategy is to have several templates for each attribute, ordered by priority.

One step up from attribute-based extraction systems are **relational extraction** systems, which deal with **multiple objects and the relations among them**. Thus, when these systems see the text “$249.99,” they need to determine not just that it is a price, but also which object has that price. A typical relational-based extraction system is FASTUS, which handles news stories about corporate mergers and acquisitions

FASTUS consists of five stages:

1. **Tokenization**: segments the stream of characters into tokens
2. **Complex-word handling** : collocations such as “set up” and “joint venture,” as well as proper names such as “Bridgestone Sports Co.”
3. **Basic-group handling** : noun groups and verb groups
4. **Complex-phrase handling** : combines the basic groups, finite-state and thus can be processed quickly
5. **Structure merging** : **merges structures** that were built up in the previous step

Disadvantage:

When information extraction must be attempted from noisy or varied input, simple finite-state approaches fare poorly. It is too hard to get all the rules and their priorities right;

Probabilistic models for information extraction

. The simplest probabilistic model for sequences with hidden state is the hidden Markov model, or HMM. HMM models a progression through a sequence of hidden states, $\mathbf{x}_t$, with an observation $\mathbf{e}_t$ at each step. The observations are the words of the text, and the hidden states are whether we are in the target, prefix, or post fix part of the attribute template, or in the background.

HMMs have two big advantages over FSAs for extraction.

- First, HMMs are probabilistic, and thus tolerant to noise. In a regular expression, if a single expected character is missing, the regex fails to match; with HMMs there is graceful degradation with missing characters/words, and we get ***a probability indicating the degree of match***, not just a Boolean match/fail.
- Second HMMs can be **trained from data**; they don't require laborious engineering of templates, and thus they can more easily be kept up to date as text changes over time.

HMM templates:

- Consist of one or more target states,
- Prefix states must precede the targets,
- Postfix states most follow the targets, and
- other states must be background.

With sufficient training data, the HMM automatically learns a structure
Once the HMMs have been learned, we can apply them to a text

Conditional random fields for information extraction

- **conditional random field**, or CRF, which models a conditional probability distribution of a set of target variables given a set of observed variables
- Like Bayesian networks, CRFs can represent many different structures of dependencies among the variables. One common structure is the **linear-chain conditional random field** for repre- senting Markov dependencies among variables in a temporal sequence.
- **Conditional Random Fields** (CRFs) are the state of the art approaches taking the sequence characteristics to do better labeling. However, as the **information** on a Web page is two-dimensionally laid out, previous linear-chain CRFs have their limitations for Web **information extraction**.
- The feature functions are the key components of CRF.
- The general form of a feature function is fi(zn−1, zn, x1:N , n), which looks at a pair of adjacent states zn−1, zn, the whole input sequence x1:N , and where we are in the sequence. The feature functions produce a real value
- Features are not limited to binary functions. Any real-valued function is allowed. Designing the features of an CRF is the most important task. In CRFs for real applications it is not uncommon to have tens of thousands or more features.
- CRF formalism gives us a great deal of flexibility in defining them. This flexibility can lead to accuracies that are higher than with less flexible models such as HMMs

Ontology extraction from large corpora

Information extraction is finding a specific set of relations (e.g., speaker, time, location) in a specific text (e.g., a talk announcement). A different applica- tion of extraction technology is building a large knowledge base or ontology of facts from a corpus.

1. First it is open-ended—we want to acquire facts about all types of domains, not just one specific domain.
2. Second, with a large corpus, this task is dominated by precision, not recall—just as with question answering on the Web.
3. Third, the results can be statistical aggregates gathered from multiple sources, rather than being extracted from one specific text.

templates:

NP **such as** NP (**,** NP)* (**,**)? ((**and** | **or**) NP)? .

NP is a variable standing for a noun phrase

This template matches the texts "diseases such as rabies affect your dog" and "supports network protocols such as DNS," concluding that rabies is a disease and DNS is a network protocol.

With a large corpus we can afford to be picky; to use only the high-precision templates. We'll miss many statements of a subcategory relationship, but most likely we'll find a paraphrase of the statement somewhere else in the corpus in a form we can use.

Automated template construction

The *subcategory* relation is so fundamental that is worthwhile to handcraft a few templates to help identify instances of it occurring in natural language text.

It is possible to *learn* templates from a few examples, then use the templates to learn more examples, from which more templates can be learned, and so on. In one of the first experiments of this kind, Brin (1999) started with a data set of just five examples:

("Isaac Asimov", "The Robots of Dawn")
("David Brin", "Startide Rising")
("James Gleick", "Chaos—Making a New Science")
("Charles Dickens", "Great Expectations") ("William Shakespeare", "The Comedy of Errors")

Clearly these are examples of the author–title relation, but the learning system had no knowl- edge of authors or titles.

Each match is defined as a tuple of seven strings,

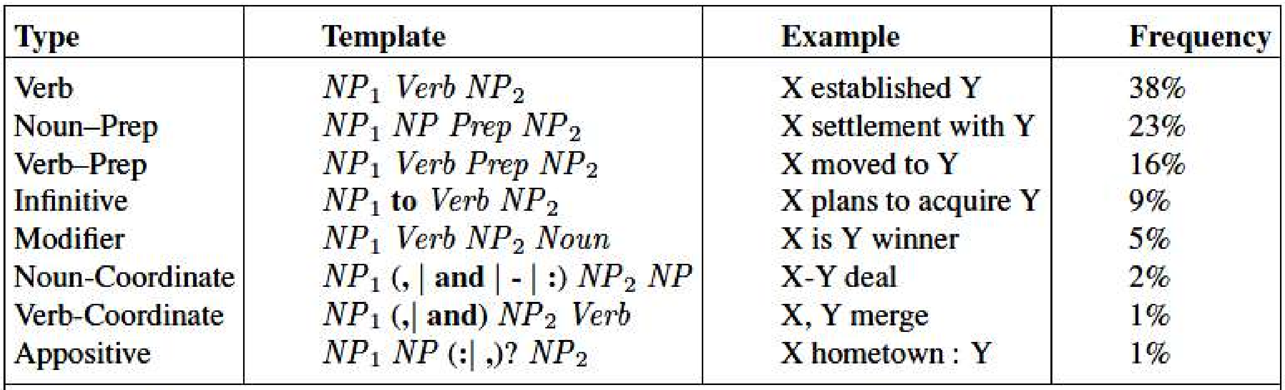
(*Author, Title, Order, Prefix, Middle, Postfix, URL*) ,

where *Order* is true if the author came first and false if the title came first, *Middle* is the characters between the author and title, *Prefix* is the 10 characters before the match, *Suffix* is the 10 characters after the match, and *URL* is the Web address where the match was made.

- Given a set of matches, a simple template-generation scheme can find templates to explain the matches.
- The biggest weakness in this approach is the sensitivity to noise. If one of the first few templates is incorrect, errors can propagate quickly.
- One way to limit this problem is to not accept a new example unless it is verified by multiple templates, and not accept a new template unless it discovers multiple examples that are also found by other templates.

Machine reading

- Automated template construction is a big step up from handcrafted template construction, but it still requires a handful of labeled examples of each relation to get started.
- To build a large ontology with many thousands of relations, even that amount of work would be onerous;
- we would like to have an extraction system with *no* human input of any kind—a system that could read on its own and build up its own database.
- Such a system would be relation-independent; would work for any relation.
- These systems work on *all* relations in parallel, because of the I/O demands of large corpora. They behave less like a traditional information- extraction system that is targeted at a few relations and more like a human reader who learns from the text itself; because of this the field has been called **machine reading**.
- Given a set of labeled examples of this type, TEXTRUNNER trains a linear-chain CRF to extract further examples from unlabeled text. The features in the CRF include function words like "to" and "of" and "the," but not nouns and verbs (and not noun phrases or verb phrases). Because TEXTRUNNER is domain-independent, it cannot rely on predefined lists of nouns and verbs.



Type	Template	Example	Frequency
Verb	NP_1 $Verb$ NP_2	X established Y	38%
Noun–Prep	NP_1 NP $Prep$ NP_2	X settlement with Y	23%
Verb–Prep	NP_1 $Verb$ $Prep$ NP_2	X moved to Y	16%
Infinitive	NP_1 **to** $Verb$ NP_2	X plans to acquire Y	9%
Modifier	NP_1 $Verb$ NP_2 $Noun$	X is Y winner	5%
Noun-Coordinate	NP_1 (, | **and** | - | :) NP_2 NP	X-Y deal	2%
Verb-Coordinate	NP_1 (,| **and**) NP_2 $Verb$	X, Y merge	1%
Appositive	NP_1 NP (:| ,)? NP_2	X hometown : Y	1%

Figure 22.3 Eight general templates that cover about 95% of the ways that relations are expressed in English.

6. NATURAL LANGUAGE PROCESSING

1. Natural Language Understanding
   - Taking some spoken/typed sentence and working out what it means
2. Natural Language Generation
   - Taking some formal representation of what you want to say and working out a way to express it in a natural (human) language (e.g., English)

Applications of NLP

- Machine Translation
- Database Access
- Information Retrieval
  - Selecting from a set of documents the ones that are relevant to a query
- Text Categorization
  - Sorting text into fixed topic categories
- Extracting data from text
  - Converting unstructured text into structured data
- Spoken language control systems
- Spelling and grammar checkers

Natural language understanding

Raw speech signal

↓ ● Speech recognition

Sequence of words spoken

↓ ● Syntactic analysis using knowledge of the grammar

Structure of the sentence

↓ ● Semantic analysis using info. about meaning of words

Partial representation of meaning of sentence

↓ ● Pragmatic analysis using info. about context

Final representation of meaning of sentence

LANGUAGE MODELS

Formal languages, such as the programming languages Java or Python, have precisely defined language models. A **language** can be defined as a set of strings; "`print(2 + 2)`" is a legal program in the language Python, whereas "`2)+(2 print`" is not. Since there are an infinite number of legal programs, they cannot be enumerated; instead they are specified by a set of rules called a **grammar**. Formal languages also have rules that define the meaning or **semantics** of a program; for example, the rules say that the "meaning" of "`2 + 2`" is 4, and the meaning of "`1/0`" is that an error is signalled

N-gram character models

Ultimately, a written text is composed of **characters**—letters, digits, punctuation, and spaces in English
A model of the probability distribution of n-letter sequences is thus called an n**-gram model**
An n-gram model is defined as a **Markov chain** of order n – 1

$$P(c_i \mid c_{1:i-1}) = P(c_i \mid c_{i-2:i-1}) \ .$$

For a trigram character model in a language with 100 characters, $\mathbf{P}(C_i|C_{i-2:i-1})$ has a million entries, and can be accurately estimated by counting character sequences in a body of text of
10 million characters or more. We call a body of text a **corpus** (plural *corpora*), from the
Latin word for *body*

language identification: given a text, determine what natural language it is written in One approach to language identification is to first build a trigram character model of each candidate language, $P(c_i \mid c_{i-2:i-1}, \ )$, where the variable ranges over languages. For each the model is built by counting trigrams in a corpus of that language.
Genre classification
Genre classification means deciding if a text is a news story, a legal document, a scientific article, etc. While many features help make this classification, counts of punctuation and other character n-gram features go a long way (Kessler *et al.*,
1997).
Named-entity recognition
Named-entity recognition is the task of finding names of things in a document and deciding what class they belong to. For example, in the text "Mr. Sopersteen was prescribed aciphex," we should recognize that "Mr. Sopersteen" is the name of a person and "aciphex" is the name of a drug. Character-level models are good for this task because they can associate the character sequence "ex " ("ex" followed by a space) with a drug name and "steen " with a person name, and thereby identify words that they have never seen before

Smoothing *n*-gram models

The major complication of n-gram models is that the training corpus provides only an esti- mate of the true probability distribution
we will adjust our language model so that sequences that have a count of zero in the training corpus will be assigned a small nonzero probability (and the other counts will be adjusted downward slightly so that the probability still sums to 1). The process od adjusting the probability of low-frequency counts is called **smoothing**.
simplest type of smoothing was suggested by Pierre-Simon Laplace in the 18th cen- tury: he said that, in the lack of further information, if a random Boolean variable X has been false in all n observations so far then the estimate for $P(X = \text{true})$ should be $1/(n+2)$.

A better approach is a **backoff model**, in which we start by estimating n-gram counts, but for
any particular sequence that has a low (or zero) count, we back off to $(n - 1)$-grams. **Linear**
interpolation smoothing is a backoff model that combines trigram, bigram, and unigram models by linear interpolation.

Model evaluation

With so many possible n-gram models—unigram, bigram, trigram, interpolated smoothing with different values of λ, etc.—how do we know what model to choose? We can evaluate a model with cross-validation Split the corpus into a training corpus and a validation corpus. Determine the parameters of the model from the training data. Then evaluate the model on the validation corpus.

The evaluation can be a task-specific metric, such as measuring accuracy on language identification. Alternatively we can have a task-independent model of language quality: cal- culate the probability assigned to the validation corpus by the model; the higher the proba- bility the better. This metric is inconvenient because the probability of a large corpus will be a very small number, and floating-point underflow becomes an issue. A different way of describing the probability of a sequence is with a measure called **perplexity**, defined as

$$\text{Perplexity}\,(c_{1:N}) = P\,(c_{1:N})\ ^{N}\ .$$

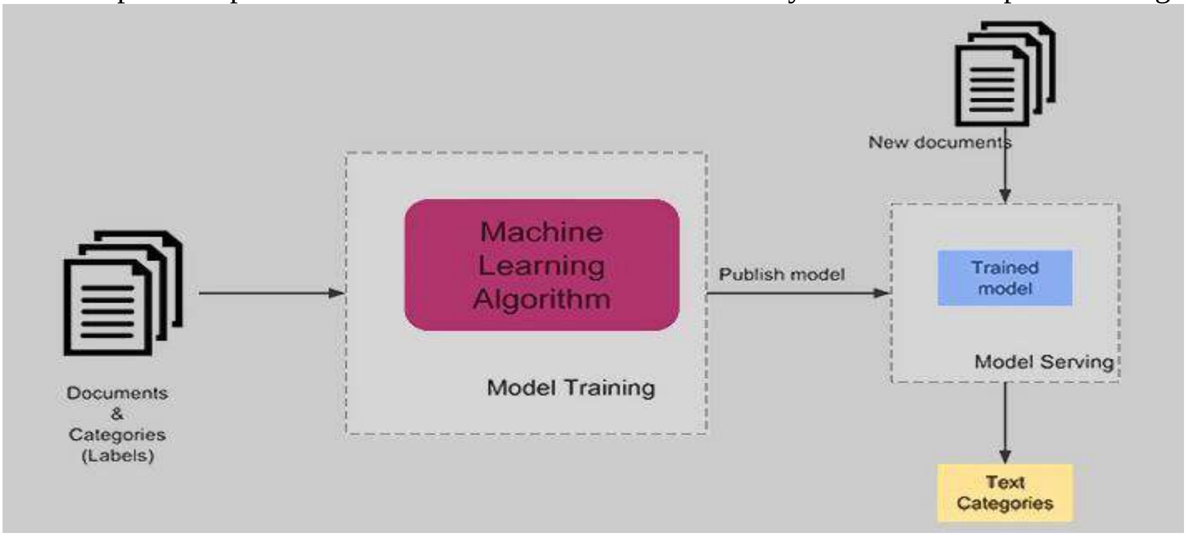
Perplexity can be thought of as the reciprocal of probability, normalized by sequence length

N-gram word models

Word n-gram models need to deal with **out of vocabulary** words With character mod- els, we didn't have to worry about someone inventing a new letter of the alphabet.[1] But with word models there is always the chance of a new word that was not seen in the training corpus, so we need to model that explicitly in our language model.

<u>Text categorization</u>

NLP techniques have proven successful in a related task: sorting text into fixed topic categories. There are several commercial services that provide access to news wire stories in this manner. A subscriber can ask for all the news on a particular industry, company, or geographic area, for example. The providers of these services have traditionally used human experts to assign



the categories. In the last few years, NLP systems have proven to be just as accurate, correctly categorizing over 90% of the news stories. They are also far faster and more consistent, so there has been a switch from humans to automated systems.
Text categorization is amenable to NLP techniques where IR is not because the categories are fixed, and thus the system builders can spend the time tuning their program to the problem. For example, in a dictionary, the primary definition of the word "crude" is vulgar, but in a large sample of the *Wall Street Journal*, "crude" refers to oil 100% of the time.

7. MACHINE TRANSLATION

Machine translation (MT) is automated translation. It is the process by which computer software is used to translate a text from one natural language (such as English) to another (such as Spanish).

To do translation well, a translator (human or machine) must read the **original text, understandthe situation to which it is referring, and find a corresponding text in the target languag**e that does a good job of describing the same or a similar situation.

Often this involves a choice. For example, the English word "you" can be translated into French as either the formal "vous" or the informal "tu." There is just no way that one can refer to the concept of "you" in French without also making a choice of formal or informal. Translators (both machine and human) sometimes find it difficult to make this choice.

To process any translation, human or automated, the meaning of a text in the original (source) language must be fully restored in the target language, i.e. the translation. While on the surface this seems straightforward, it is far more complex.

Translation is not a mere word-for-word substitution. A translator must interpret and analyze all of the elements in the text and know how each word may influence another. This requires extensive expertise in grammar, syntax (sentence structure), semantics (meanings), etc., in the source and target languages, as well as familiarity with each local region.

Human and machine translation each have their share of challenges. For example, no two individual translators can produce identical translations of the same text in the same language pair, and it may take several rounds of revisions to meet customer satisfaction. But the greater challenge lies in how machine translation can produce publishable quality translations.

A representation language that makes all the distinctions necessary for a set of languages is called an **interlingua.** Some systems attempt to analyze the source language text all the way into an interlingua knowledge representation and then generate sentences in the target language from that representation.

This is difficult because it involves three unsolved problems:

1. Creating a complete knowledge representation of everything;
2. Parsing into that representation; and
3. Generating sentences from that representation.

Rule-Based Machine Translation Technology

→ Rule-based machine translation relies on countless built-in linguistic rules and millions of bilingual dictionaries for each language pair.
→ The software parses text and creates a transitional representation from which the text in the target language is generated.
→ This process requires extensive lexicons with morphological, syntactic, and semantic information, and large sets of rules. The software uses these complex rule sets and then transfers the grammatical structure of the source language into the target language.
→ Translations are built on gigantic dictionaries and sophisticated linguistic rules. Users can improve the out-of-the-box translation quality by adding their terminology into the translation process. They create user-defined dictionaries which override the system's default settings.

In most cases, there are two steps:

→ An initial investment that *significantly increases the quality* at a limited cost, and
→ An *ongoing investment to increase quality incrementally*.
→ While rule-based MT brings companies to the quality threshold and beyond, the quality improvement process may be long and expensive.

Statistical Machine Translation Technology

Statistical machine translation utilizes statistical translation models whose parameters stem from the **analysis of monolingual and bilingual corpora**.

Building statistical translation models is a quick process, but **the technology relies heavily on existing multilingual corpora**.

A minimum of 2 million words for a specific domain and even more for general language are required. Theoretically it is possible to reach the quality threshold but most companies do not have such large amounts of existing multilingual corpora to build the necessary translation models.

Additionally, statistical machine translation is CPU intensive and requires an extensive hardware configuration to run translation models for average performance levels.

Rule-Based MT vs. Statistical MT

→ Rule-based MT provides *good out-of-domain quality* and is by nature predictable.
→ Dictionary-based customization guarantees improved quality and compliance with corporate terminology. But translation results may lack the fluency readers expect.
→ In terms of investment, the customization cycle needed to reach the quality threshold can be long and costly. The performance is high even on standard hardware.

Statistical MT provides good quality when large and qualified corpora are available. The translation is fluent, meaning it reads well and therefore meets user expectations. However, the translation is neither predictable nor consistent. Training from good corpora is automated and cheaper. But training on general language corpora, meaning text other than the specified domain, is poor. Furthermore, statistical MT requires significant hardware to build and manage large translation models.

Transfer model
They keep a database of translation rules (or examples), and whenever the rule (or example) matches, they translate directly.
Transfer can occur at the lexical, syntactic, or semantic level.
For example, a strictly syntactic rule maps English [*Adjective Noun*] to French [*Noun Adjective*]. A mixed syntactic and lexical rule maps French [S_1 "et puis" S_2] to English [S_1 "and then" S_2].

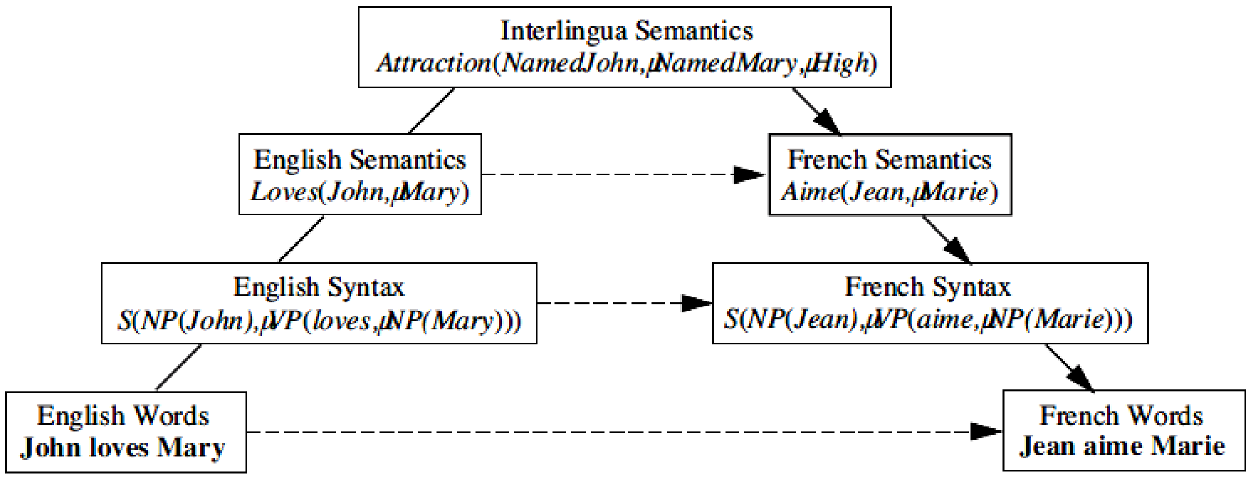


Figure 23.12 The Vauquois triangle: schematic diagram of the choices for a machine translation system (Vauquois, 1968). We start with English text at the top. An interlingua-based system follows the solid lines, parsing English first into a syntactic form, then into a semantic representation and an interlingua representation, and then through generation to a semantic, syntactic, and lexical form in French. A transfer-based system uses the dashed lines as a shortcut. Different systems make the transfer at different points; some make it at multiple points.

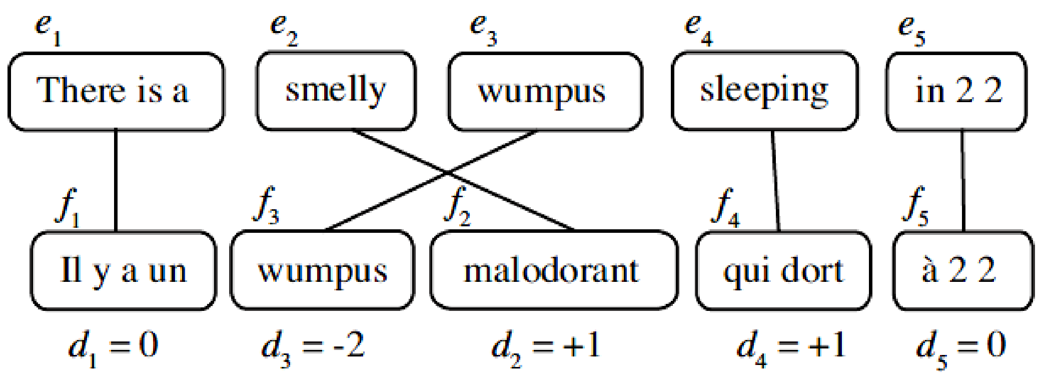


Figure 23.13 Candidate French phrases for each phrase of an English sentence, with distortion (d) values for each French phrase.

All that remains is to learn the phrasal and distortion probabilities

1. Find parallel texts
2. Segment into sentences
3. Align sentences
4. Align phrases
5. Extract distortions
6. Improve estimates with EM

EXAMPLE

TAUM-METEO system, developed by the University of Montreal, which translates weather reports from English to French. It works because the language used in these government weather reports is highly stylized and regular.

A representative system is SPANAM (Vasconcellos and Leon, 1985), which can translate a Spanish passage into English of this quality.

→ This is mostly understandable, but not always grammatical and rarely fluent.

→ Another possibility is to invest the human effort on pre-editing the original document.
→ If the original document can be **made to conform to a restricted subset of English** (or whatever the original language is), then it can sometimes be translated without the need for post-editing.
→ This approach is particularly cost-effective when there is a need to translate one document into many languages, as is the case for legal documents

The problem is that different languages categorize the world differently. A majority of the situations that are covered by the English word "open" are also covered by the German word "offen," but the boundaries of the category differ across languages.
In English, we extend the basic meaning of "open" to cover open markets, open questions, and open job offerings.
In German, the extensions are different. Job offerings are "freie," not open, but the concepts of loose ice, private firms, and blank checks all use a form of "offen."

8. DATABASE ACCESS
The first major success for natural language processing (NLP) was in the area of database access.
The disadvantage is that the user never knows which wordings of a query will succeed and which are outside the system's competence some commercial systems have built up large enough grammars and lexicons to handle a fairly wide variety of inputs. The main challenge for current systems is to follow the context of an interaction. The user should be able to ask a series of questions where some of them implicitly refer to earlier questions or answers:

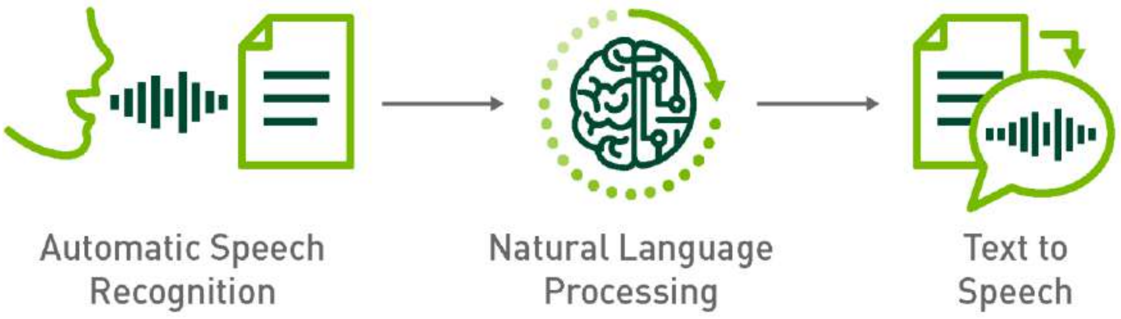
What countries are north of the equator?
How about south?
Show only the ones outside Australasia.
What is their total area?
Some systems (e.g., TEAM (Grosz *et al.,* 1987)) handle problems like this to a limited degree,

Natural Language Inc. and Symantec are still selling database access tools that use natural language, but customers are less likely to make their buying decisions based on the strength of the natural language component than on the graphical user interface or the degree of integration of the database with spreadsheets and word processing.
Natural language is not always the most natural way to communicate: sometimes it is easier to point and click with a mouse to express an idea.
The emphasis in practical NLP has now shifted away from database access to the broad **interpretation.**

In part, this is a reflection of a change in the computer industry.
In the early 1980s, most online information was stored in databases or spreadsheets. Now the majority of online information is text: email, news, journal articles, reports, books, encyclopedias. Most computer users find there is too much information available, and not enough time to sort through it. Text interpretation programs help to retrieve, categorize, filter, and extract information from text. Text interpretation systems can be split into three types: information retrieval, text categorization, and data extraction.

9. SPEECH RECOGNITION
It is the task of mapping from a digitally encoded acoustic signal to a string of words. **Speech understanding** is the task of mapping from the acoustic signal all the way to an interpretation of the meaning of the utterance.
A speech understanding system must answer three questions:

1. What speech sounds did the speaker utter?
2. What words did the speaker intend to express with those speech sounds?
3. What meaning did the speaker intend to express with those words?



To answer question 1, we have to first decide what a speech sound is. It turns out that all human languages use a limited repertoire of about 40 or 50 sounds, called **phones.** Roughly speaking, a phone is the sound that corresponds to a single vowel or consonant, but there are some complications:

Combinations of letters such as "th" and "ng" produce single phones, and some letters produce different phones in different contexts (for example, the "a" in *rat* and *rate.* Once we know what the possible sounds are, we need to characterize them in terms of features that we can pick out of the acoustic signal, such as the frequency or amplitude of the sound waves.

Question 2 is conceptually much simpler. You can think of it as looking up words in a dictionary that is arranged by pronunciation.
We get a sequence of three phones, *[k], [ce],* and *ft],* and find in the dictionary that this is the pronunciation for the word "cat."

Two things make this difficult.
The first is the existence of **homophones,** different words that sound the same, like "two" and "too."1
The second is **segmentation,** the problem of deciding where one word ends and the next begins.

Question 3 we already know how to answer—use the parsing and analysis algorithms
Some speech understanding systems extract the most likely string of words and pass them directly to an analyzer. Other systems have a more complex control structure that considers multiple possible word interpretations so that understanding can be achieved even if some individual words are not recognized correctly.

<u>Signal processing</u>
Sound is an analog energy source. When a sound wave strikes a microphone, it is converted to an electrical current, which can be passed to an analog-to-digital converter to yield a stream of bits representing the sound.

We have two choices in deciding how many bits to keep.
First, the **sampling rate** is the frequency with which we look at the signal. For speech, a sampling rate between 8 and 16 KHz (i.e., 8 to 16,000 times per second) is typical. Telephones deliver only about 3 KHz.
Second, the **quantization factor** determines the precision to which the energy at each sampling point is recorded. Speech recognizers typically keep 8 to 12 bits. That means that a low-end system, sampling at 8 KHz with 8-bit quantization, would require nearly half a megabyte per minute of speech.

The first step in coming up with a better representation for the signal is to group the samples together into larger blocks called ***frames.*** This makes it possible to analyze the whole frame for the appearance of speech phenomena such as a rise or drop in frequency, or a sudden onset or cessation of energy. Within each frame, we represent what is happening with a vector **of features.**
Figure 24.33 shows frames with a vector of three features. Note that the frames overlap; this prevents us from losing information if an important acoustic event just happens to fall on a frame boundary.

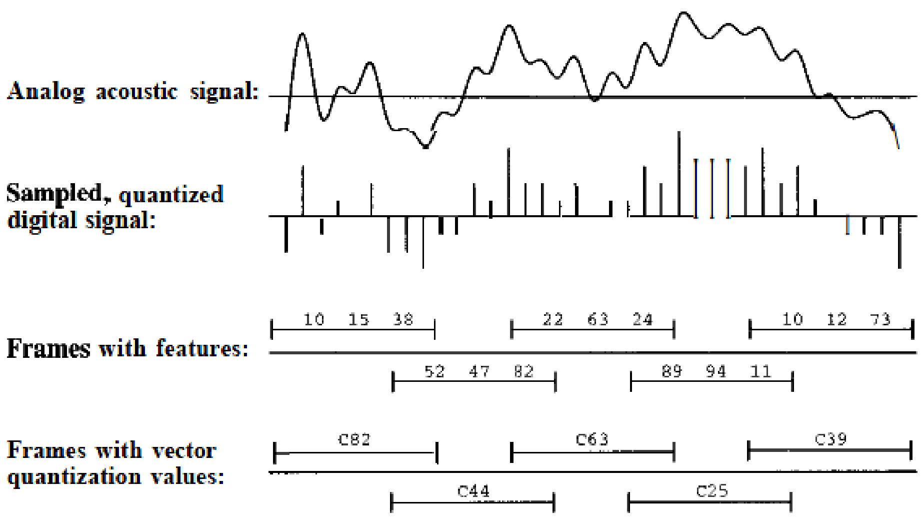


Figure 24.33 Translating the acoustic signal into a sequence of vector quantization values. (Don't try to figure out the numbers; they were assigned arbitrarily.)

The final step in many speech signal processing systems is **vector quantization.** If there are *n* features in a frame, we can think of this as an n-dimensional space containing many points.
Some information is lost in going from a feature vector to a label that summarizes a whole neighborhood around the vector, but there are automated methods for choosing an optimal quantization of the feature vector space so that little or no inaccuracy is introduced (Jelinek, 1990).

First, we end up with a representation of the speech signal that is compact. But more importantly, we have a representation that is likely to encode features of the signal that will be useful for word recognition. A given speech sound can be pronounced so many ways: loud or soft, fast or slow, high-pitched or low, against a background of silence or noise, and by any of millions of different speakers each with different accents and vocal tracts. Signal processing hopes to capture enough of the important features so that the commonalities that define the sound can be picked out from this backdrop of variation.
The dual problem, **speaker identification,** requires one to focus on the variation instead of the commonalities in order to decide who is speaking.

Defining the overall speech recognition model
Speech recognition is the diagnostic task of recovering the words that produce a given acoustic signal. It is a classic example of reasoning with uncertainty. We are uncertain about how well the microphones (and digitization hardware) have captured the actual sounds, we are uncertain about which phones would give rise to the signal, and we are uncertain about which words would give rise to the phones. As is often the case, the diagnostic task can best be approached with a causal model—the words cause the signal.

We can break this into components with Bayes' rule:

$$P(words|signal) = \frac{P(words)P(signal|words)}{P(signal)}$$

Given a *signal*, our task is to find the sequence of *words* that maximizes *P(words\signal).* Of the three components on the right-hand side, *P(signal)* is a normalizing constant that we can ignore. *P(words)* is known as the **language model.** It is what tells us, when we are not sure if we heard

"bad boy" or "pad boy" that the former is more likely. Finally, *P(signal\words)* is the **acoustic model.** It is what tells us that "cat" is very likely to be pronounced *[kcet].*

<u>The language model: P(words)</u>

Assign a probability to each of the (possibly infinite) number of strings. Context-free grammars are no help for this task, but probabilistic context-free grammars (PCFGs) are promising.
Bigram model : it says that the probability of any given word is determined solely by the previous word in the string

Trigrammodel that provides values for />(w;|w,-_iw/_2). This is a more powerful language model, capable of determining that "ate a banana" is more likely than "ate a bandana." The problem is that there are so many more parameters in trigram models that it is hard to get enough training data to come up with accurate probability estimates.

The acoustic model: P(signallwords)

The acoustic model is responsible for saying what sounds will be produced when a given string of words is uttered. We divide the model into two parts. First, we show how each word is described as a sequence of phones, and then we show how each phone relates to the vector quantization values extracted from the acoustic signal.
Markov model
Two sources of phonetic
First, different dialects have different pronunciations.
The top of Figure. gives an example of this: for "tomato," you say [tow mey tow] and I say [tow maa tow]. The alternative pronunciations are specified as a **Markov model.**

In general, a **Markov model is a way of describing a process that goes through a series of states**. The model describes all the possible paths through the state space and assigns a probability to each one. The probability of transitioning from the current state to another one depends only on the current state, not on any prior part of the path.

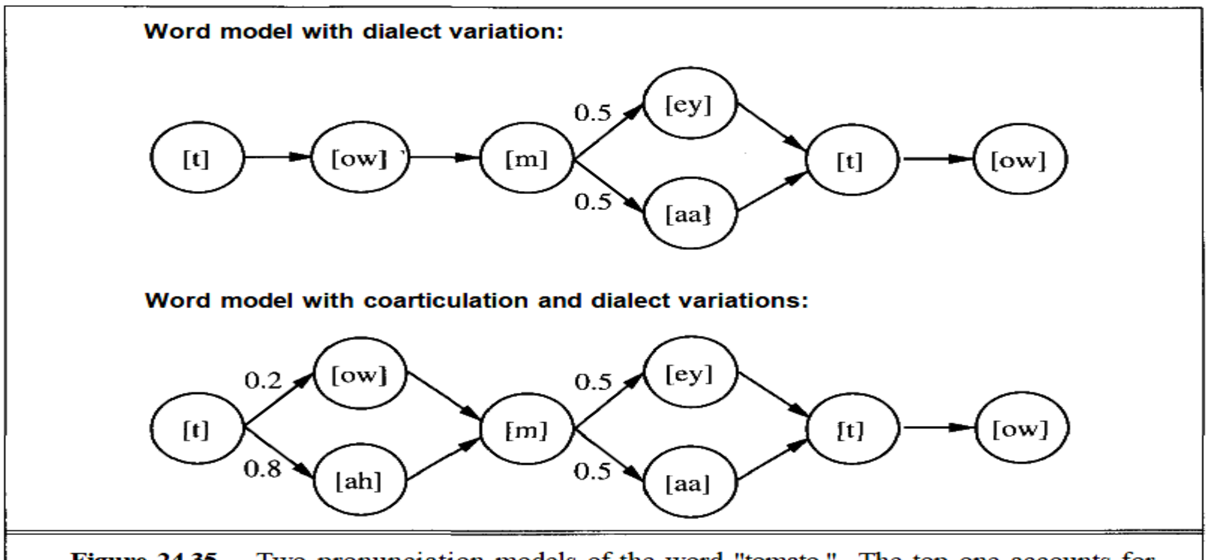

Word model with dialect variation:
[t]
[ow]
[m]
0.5
[ey]
0.5
[aa]
[t]
[ow]
Word model with coarticulation and dialect variations:
[t]
0.2
[ow]
0.8
[ah]
[m]
0.5
[ey]
0.5
[aa]
[t]
[ow]

Figure 24.35 Two pronunciation models of the word "tomato." The top one accounts for dialect differences. The bottom one does that and also accounts for a coarticulation effect.

Markov model with seven states (circles), each corresponding to the production of a phone. The arrows denote allowable transitions between states, and each transition has a probability associated with it.3 There are only two possible paths through the model, one corresponding to the phone sequence [t ow m ey t ow] and the other to [t ow m aa tow]. The probability of a path is the product of the probabilities on the arcs that make up the path. In this case, most of the arc probabilities are 1 and we have

P([towmeytow] ("tomato") = P([r«w»iaa?ow] ("tomato") = 0.5

$$P([towmeytow] | \text{"tomato"}) = P([towmaatow] | \text{"tomato"}) = 0.5$$

*The second source of phonetic variation is **coarticulation.***

Similar models would be constructed for every word we want to be able to recognize. Now if the speech signal were a list of phones, then we would be done with the acoustic model. We could take a given input signal (e.g., [towmeytow]) and compute *P(signal\words)* for various word strings (e.g., "tomato," "toe may tow," and so on). We could then combine these with *P(words)* values taken from the language model to arrive at the *words* that maximize *P(words\signal).*

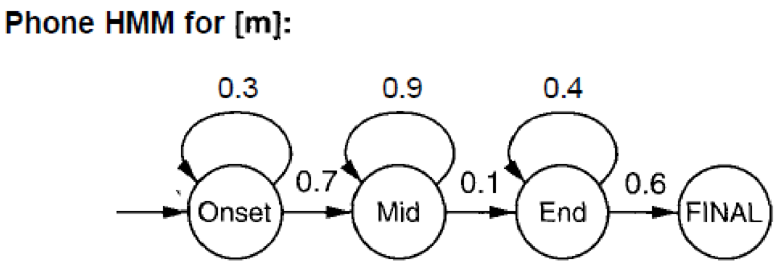
hidden Markov model or **HMM**

Unfortunately, signal processing does not give us a string of phones. So all we can do so far is maximize *P(words\phones).* Figure 24.36 shows how we can compute *P(signal\phone)* using a model called a **hidden Markov model** or **HMM.** The model is for a particular phone,[m], but all phones will have models with similar topology.

A hidden Markov model is just like a regular Markov model in that it describes a process that goes through a *sequence of states*. The difference is that in a regular Markov model, the output is a sequence of state names, and because each state has a unique name, the output uniquely determines the path through the model.

In a hidden Markov model, each state has *a probability distribution of possible outputs*, and the same output can appear in more than one state.4

HMMs are called hidden models because the true state of the model is hidden from the observer. In general, when you see that an HMM outputs some symbol, you can't be sure what state the symbol came from.



Output probabilities for the phone HMM:

Onset:	Mid:	End:
C1: 0.5	C3: 0.2	C4: 0.1
C2: 0.2	C4: 0.7	C6: 0.5
C3: 0.3	C5: 0.1	C7: 0.4

Figure 24.36 An HMM for the phone [m]. Each state has several possible outputs, each with its own probability.

Actually, most phones have a duration of 50-100 milliseconds, or 5-10 frames at 10 msec/frame. So the [C1,C4,C6J sequence is unusually quick. Suppose we have a more typical speaker who generates the sequence [C1,C1,C4,C4,C6,C6J while producing the phone. It turns out there are two paths through the model that generate this sequence. In one of them both C4s come from the Mid state (note the arcs that loop back), and in the other the second C4 comes from the End state. We calculate the probability that this sequence came from the [m] model in the same way: take the sum over all possible paths of the probability of the path times the probability that the path generates the sequence.

P([Cl,C\,C4, C4, C6, C6]jLm]) =
(0.3 x 0.7 x 0.9 x 0.1 x 0.4 x 0.6) x (0.5 x 0.5 x 0.7 x 0.7 x 0.5 x 0.5) +
(0.3 x 0.7 x 0.1 x 0.4 x 0.4 x 0.6) x (0.5 x 0.5 x 0.7 x 0.1 x 0.5 x 0.5)
= 0.0001477

Putting the models together

We have described three models.

The language bigram model gives us *P(wordi\wordi-\}.*

The word pronunciation HMM gives us *P(phones\word).*

The phone HMM gives us *P(signal\phone).*

If we want to compute *P(words\signal),* we will need to combine these models in some way. Oneapproach is to combine them all into one big HMM. The bigram model can be thought of as an HMM in which every state corresponds to a word and every word has a transition arc to every other word. Now replace each word-state with the appropriate word model, yielding a bigger model in which each state corresponds to a phone. Finally, replace each phone-state with the appropriate phone model, yielding an even bigger model in which each state corresponds to a distribution of vector quantization values.

Some speech recognition systems complicate the picture by dealing with coarticulation effects at either the word/word or phone/phone level. For example, we could use one phone model for [ow] when it follows a [t] and a different model for [ow] when it follows a [g]. There are many trade-offs to be made—a more complex model can handle subtle effects, but it will be harder to train. Regardless of the details, we end up with one big HMM that can be used to compute*P(words\signal).*

10. ROBOT (HARDWARE –PERCEPTION –PLANNING –MOVING)

Robots are physical agents that perform ROBOT tasks by manipulating the physical world. To do so they are equipped with **effectors** such as legs, wheels, joints, and grippers Effectors have a single purpose: to assert physical forces on the environment.

Robots are also equipped with **sensors**, which allow them to perceive their environment. Present day robotics employs a diverse set of sensors, including cameras and lasers to measure the environment, and gyroscopes and accelerometers to measure the robot’s own motion.

Manipulators, or robot arms are physically anchored to their workplace, for example in a factory assembly line or on the International Space Station.

Mobile robot

Mobile robots move about their environment using wheels, legs, or similar mechanisms. They have been put to use delivering food in hospitals, moving containers at loading docks, and similar tasks. **Unmanned ground vehicles**, or UGVs, drive autonomously on streets, highways, and off-road. The **planetary rover** explored Mars for a period of 3 months in 1997. Other types of mobile robots include **unmanned air vehicles** (UAVs), commonly used for surveillance, crop-spraying,

Autonomous underwater vehicles (AUVs) are used in deep sea exploration. Mobile robots deliver packages in the workplace and vacuum the floors at home.

The third type of robot combines mobility with manipulation, and is often called a **mobile manipulator**. **Humanoid robots** mimic the human torso.

The field of robotics also includes prosthetic devices (artificial limbs, ears, and eyes for humans), intelligent environments (such as an entire house that is equipped with sensors and effectors), and multibody systems, wherein robotic action is achieved through swarms of small cooperating robots. Real robots must cope with environments that are partially observable, stochastic, dynamic, and continuous.

Robotics brings together many of the concepts, including probabilistic state estimation, perception, planning, unsupervised learning, and reinforcement learning.

ROBOT HARDWARE

Sensors
Sensors are the perceptual interface between robot and environment.
Passive sensors, such as cameras, are true observers of the environment: they capture signals that are generated by other sources in the environment.
Active sensors, such as sonar, send energy into the environment.

They rely on the fact that this energy is reflected back to the sensor. Active sensors tend to provide more information than passive sensors, **Range finders** are sensors that measure the distance to nearby objects.

Sonar sensors emit directional sound waves, which are reflected by objects, with some of the sound making it back into the sensor. The time and intensity of the returning signal indicates the distance to nearby objects.

Sonar is the technology of choice for autonomous underwater vehicles. **Stereo vision** (see Section 24.4.2) relies on multiple cameras to image the environment from slightly different viewpoints, analyzing the resulting parallax in these images to compute the range of surrounding objects. For mobile ground robots, sonar and stereo vision are now rarely used, because they are not reliably accurate.

Time of flight camera :Most ground robots are now equipped with optical range finders. Just like sonar sensors, optical range sensors emit active signals (light) and measure the time until a reflection

Scanning lidars tend to provide longer ranges than time of flight cameras, and tend to perform better in bright daylight.

End of range sensing are **tactile sensors** such as whiskers, bump panels, and touch-sensitive skin. These sensors measure range based on physical contact, and can be deployed only for sensing objects very close to the robot.
location sensors
Most location sensors use range sensing as a primary component to determine location. Outdoors, the **Global Positioning System** (GPS) is the most common solution to the localization problem. GPS the distance to satellites that emit pulsed signals. At present, there are 31 satellites in orbit, transmitting signals on multiple frequencies. GPS receivers can recover the distance to these satellites by analyzing phase shifts. By triangulating signals from multiple satellites, GPS receivers can determine their absolute location on Earth to within a few meters.

Differential GPS involves a second ground receiver with known location, providing millimeter accuracy under ideal conditions. Unfortunately, GPS does not work indoors or underwater.

Proprioceptive sensors, which inform the robot of its own motion. To measure the exact configuration of a robotic joint, motors are often equipped with **shaft decoders** that count the revolution of motors in small increments. On mobile robots, shaft decoders that report wheel revolutions can be used for **odometry**—the measurement of distance traveled.

Inertial sensors, such as gyroscopes, rely on the resistance of mass to the change of velocity. They can help reduce uncertainty.

Other important aspects of robot state are measured by **force sensors** and **torque sensors**. These are indispensable when robots handle fragile objects or objects whose exact shape and location is unknown. Imagine a one-ton robotic manipulator screwing in a light bulb. It would be all too easy to apply too much force and break the bulb. Force sensors allow the robot to sense how hard it is gripping the bulb, and torque sensors allow it to sense how hard it is turning. Good sensors can measure forces in all three translational and three rotational directions. They do this at a frequency of several hundred times a second, so that a robot can quickly detect unexpected forces and correct its actions before it breaks a light bulb.

Effectors
Effectors are the means by which robots move and change the shape of their bodies. To understand the design of effectors, it will help to talk about motion and shape in the abstract, using the concept of a **degree of freedom** (DOF)

- For nonrigid bodies, there are additional degrees of freedom within the robot itself.
- Robot joints also have one, two, or three degrees of freedom each.
- Six degrees of freedom are required to place an object, such as a hand, at a particular point in a particular orientation.

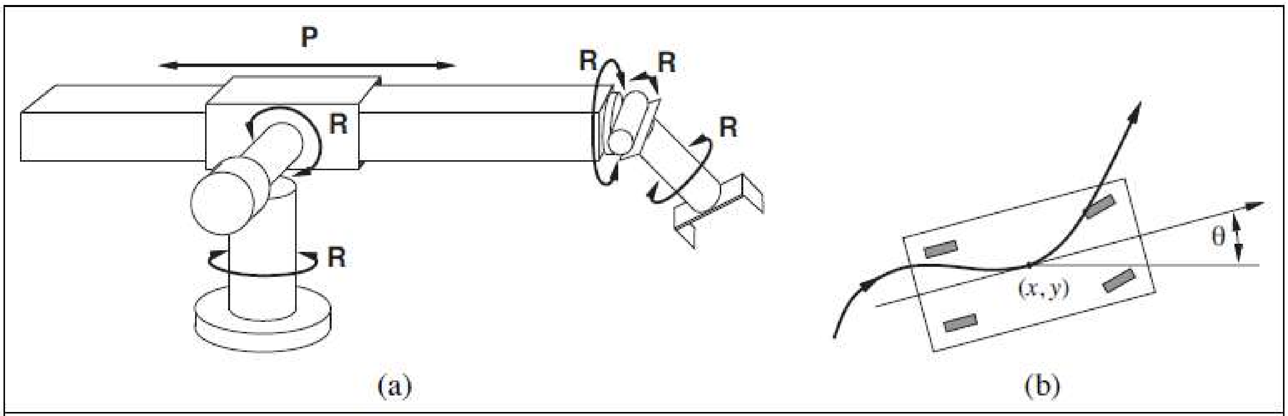


Figure 25.4 (a) The Stanford Manipulator, an early robot arm with five revolute joints (R) and one prismatic joint (P), for a total of six degrees of freedom. (b) Motion of a nonholonomic four-wheeled vehicle with front-wheel steering.

the car has three **effective degrees of freedom** but two **control** DOF **lable degrees of freedom**. **Differential drive** robots possess two independently actuated wheels (or tracks), one on each side, as on a military tank.

Legs, unlike wheels, can handle rough terrain. However, legs are notoriously slow on flat surfaces, and they are mechanically difficult to build.

A robot that can remain STABLE upright without moving its legs is called **statically stable**. A robot is statically stable if its center of gravity is above the polygon spanned by its legs.

C omplete robot also needs a source of power to drive its effectors. The **electric motor** is the most popular mechanism for both manipulator actuation and locomotion, but **pneumatic actuation** using compressed gas and **hydraulic actuation** using pressurized fluids also have their application niches.

11. ROBOTIC PERCEPTION
Perception is the process by which robots map sensor measurements into internal representations of the environment. Perception is difficult because sensors are noisy, and the environment is partially observable, unpredictable, and often dynamic.

The robot's own past actions as observed variables in the model. Figure 25.7 shows the notation used in this chapter: **X**t is the state of the environment (including the robot) at time t, **Z**t is the observation received at time t, and At is the action taken after the observation is received.

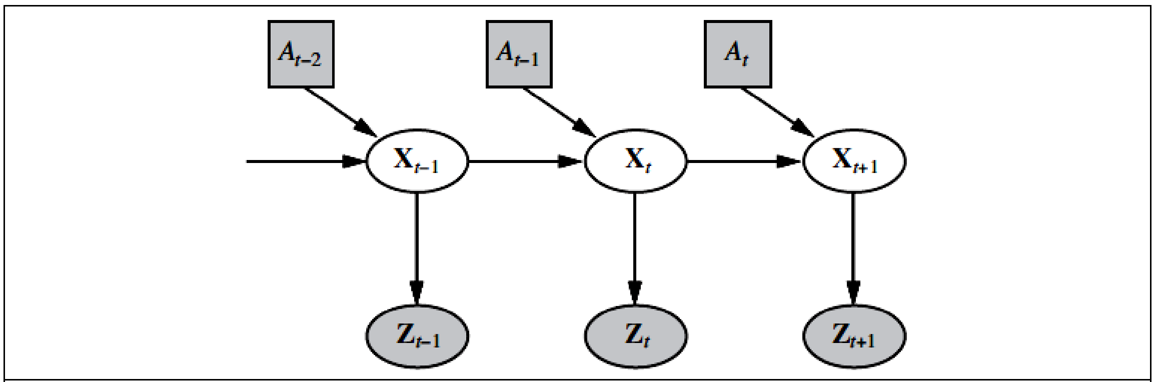


Figure 25.7 Robot perception can be viewed as temporal inference from sequences of actions and measurements, as illustrated by this dynamic Bayes network.

We would like to compute the new belief state, **P**(**X**t+1 | **z**1:t+1, a1:t), from the current belief state **P**(**X**t | **z**1:t, a1:t−1) and the new observation **z**t+1.
we condition explicitly on the actions as well as the observations, and we deal with *continuous* rather than *discrete* variables.

The probability **P**(**X**t+1 | **x**t, at) : is the **transition model** or **motion model**, and **P**(**z**t+1 | **X**MOTION MODEL t+1) : is the **sensor model**.

<u>Localization and mapping</u>
Localization is the problem of finding out where things are—including the robot itself. Knowledge about where things are is at the core of any successful physical interaction with the environment. For example, robot manipulators must know the location of objects they seek to manipulate; navigating robots must know where they are to find their way around.

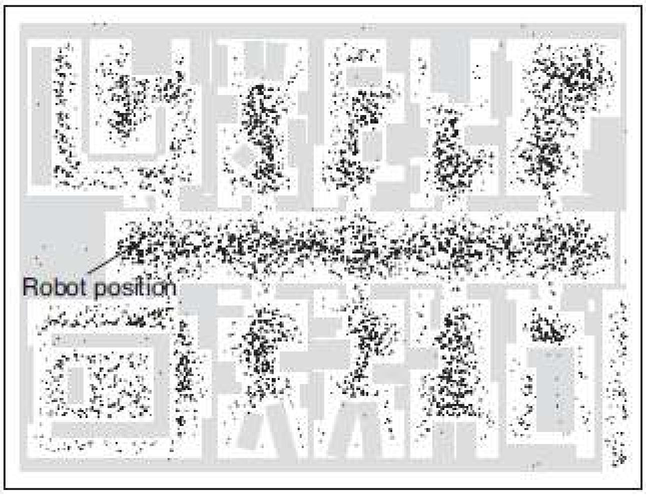
<u>Monte Carlo localization</u>
Localization using particle filtering is called **Monte Carlo localization**, or MCL. The MCL algorithm is an instance of the particle-filtering algorithm. All we need to do is supply the appropriate motion model and sensor model.

The operation of the algorithm is illustrated in as the robot finds out where it is inside an office building. In the first image, the particles are uniformly distributed based on the prior, indicating global uncertainty about the robot's position. In the second image, the first set of measurements arrives and the particles form clusters in the areas of high posterior belief. In the third, enough measurements are available to push all the particles to a single location.

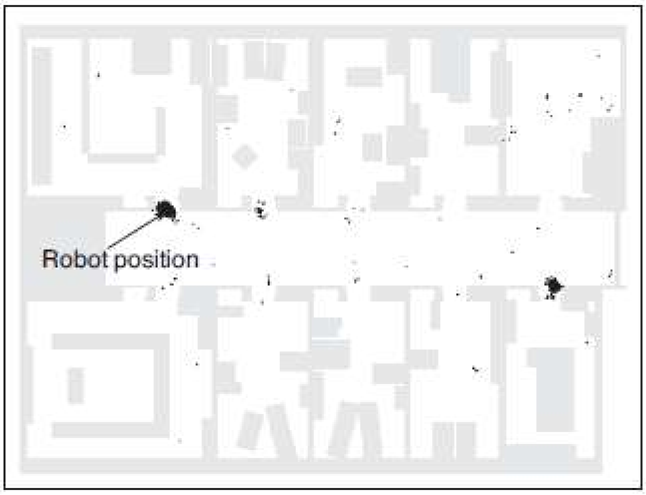
<u>Kalman filter</u>
The Kalman filter is the other major way to localize. A Kalman filter represents the posterior **P**(**X**t | **z**1:t, a1:t−1) by a Gaussian. The mean of this Gaussian will be denoted **μ** t and its covariance **Σ**t. The main problem with Gaussian beliefs is that they are only closed under linear motion models f and linear measurement models h. localization algorithms using the Kalman LINEARIZATION filter **linearize** the motion and sensor models.

The problem of needing to know the identity of landmarks is an instance of the **data association** problem

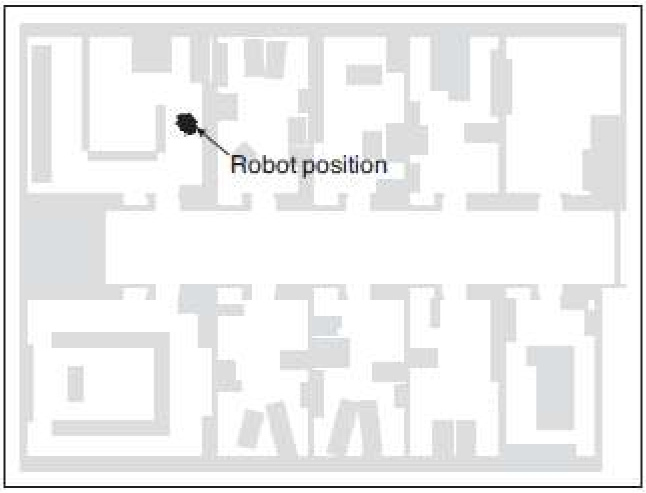
In some situations, no map of the environment is available. Then the robot will have to acquire a map. This is a bit of a chicken-and-egg problem: the navigating robot will have to determine its location relative to a map it doesn't quite know, at the same time building this map while it doesn't quite know its actual location. This problem is important for many robot applications, and it has been studied extensively under the name **simultaneous localization and mapping**, abbreviated as **SLAM**.



Monte Carlo localization, a particle filtering algorithm for mobile robot localization.
(a) Initial, global uncertainty.



(b) Approximately bimodal uncertainty after navigating
in the (symmetric) corridor.



(c) Unimodal uncertainty after entering a room and finding it to be distinctive.

Machine learning in robot perception
Machine learning plays an important role in robot perception. One common approach is to map high dimensional sensor streams into lower-dimensional spaces using unsupervised machine learning methods

Another machine learning technique enables robots to continuously adapt to broad changes in sensor measurements. Adaptive perception techniques enable robots to adjust to such changes.

Methods that make robots collect their own training data (with labels!) are called **selfsupervised**. In this instance, the robot SELF-SUPERVISED uses machine learning to leverage a short-range sensor that works well for terrain classification into a sensor that can see much farther.

12. ROBOT : MOVING
All of a robot's deliberations ultimately come down to deciding how to move effectors. The **point-to-point motion** problem is to deliver the robot or its end effector to a designated target location.

We begin by finding a suitable representation in which motion-planning problems can be described and solved. It turns out that the **configuration space**—the space of robot states defined by location, orientation, and joint angles—is a better place to work than the original PATH PLANNING 3D space. The **path planning** problem is to find a path from one configuration to another in configuration space.

There are two main approaches: **cell decomposition** and **skeletonization**.

Configuration space
This raises the question of how to map between workspace coordinates and configuration space. Transforming configuration space coordinates into workspace coordinates is simple: it involves a series of straightforward coordinate transformations. These transformations are linear for prismatic joints and trigonometric for revolute joints. This chain of coordinate transformation is known as **kinematics**.

The inverse problem of calculating the configuration of a robot whose effector location is specified in workspace coordinates is known as **inverse kinematics**. The second problem with configuration space representations arises from the obstacles that may exist in the robot's workspace

Cell decomposition methods
first CELL approach to path planning uses **cell decomposition**—that is, it decomposes the free space into a finite number of contiguous regions, called cells. The path-planning problem then becomes a discrete graph-search problem, very much like the search problems Such a decomposition has the advantage that it is extremely simple to implement, but it also suffers from three limitations.

First, it is workable only for low-dimensional configuration spaces,
Second, there is the problem of what to do with cells that are "mixed"—that is, neither entirely within free space nor entirely within occupied space.
Third, any path through a discretized state space will not be smooth. It is generally difficult to guarantee that a smooth solution exists near the discrete path.

Skeletonization methods
The second major family of path-planning algorithms is based on the idea of **skeletonization**. These algorithms reduce the robot's free space to a one-dimensional representation, for which the planning

problem is easier. This lower-dimensional representation is called a **skeleton** of the configuration space.

It is easy to show that this can always be achieved by a straight-line motion in configuration space.
it is a **Voronoi graph** of the free space—the set of all points that are equidistant to two or more obstacles. To do path planning with a Voronoi graph, the robot first changes its present configuration to a point on the Voronoi graph. It is easy to show that this can always be achieved by a straight-line motion in configuration space. Second, the robot follows the Voronoi graph until it reaches the point nearest to the target configuration. Finally, the robot leaves the Voronoi graph and moves to the target. Again, this final step involves straight-line motion in configuration space.

An alternative to the Voronoi graphs is the **probabilistic roadmap**, a skeletonization approach that offers more possible routes, and thus deals better with wide-open spaces.

13. PLANNING UNCERTAIN MOVEMENTS

None of the robot motion-planning algorithms discussed thus far addresses a key characteristic of robotics problems: *uncertainty*. In robotics, uncertainty arises from partial observability of the environment and from the stochastic (or unmodeled) effects of the robot's actions.

The field of robotics has adopted a range of techniques for accommodating uncertainty. If the robot faces uncertainty only in its state transition, but its state is fully observable, the problem is best modeled as a Markov decision process (MDP). The solution of an MDP is an optimal **policy**, which tells the robot what to do in every possible state.

For example, the **coastal navigation** heuristic requires the robot to stay near known landmarks to decrease its uncertainty. Another approach applies variants of the probabilistic roadmap planning method to the belief space representation. Such methods tend to scale better to large discrete POMDPs.

Robust methods

Uncertainty can also be handled using so-called **robust control** methods rather than probabilistic methods. A robust method is one that assumes a *bounded* amount of uncertainty in each aspect of a problem, but does not assign probabilities to values within the allowed interval. A robust solution is one that works no matter what actual values occur, provided they are within the assumed interval.

Fine-motion planning involves moving a robot arm in very close proximity to a static environment object.

A fine-motion plan consists of a series of **guarded motions**. The termination conditions are contact with a surface. To model uncertainty in control, we assume that instead of moving in the commanded direction, the robot's actual motion lies in the cone Cv about it.

Dynamics and control

The transition model for a dynamic state representation includes the effect of forces on this rate of change. the dynamic state has higher dimension than the kinematic space, and the curse of dimensionality would render many motion planning algorithms inapplicable for all but the most simple robots. For this reason, practical robot system often rely on simpler kinematic path planners.

A common technique to compensate for the limitations of kinematic plans is to use a CONTROLLER separate mechanism, a **controller**, for keeping the robot on track.

Potential-field control

potential fields as an additional cost function in robot motion planning, but they can also be used for generating robot motion directly, dispensing with the path planning phase altogether.

Define an attractive force that pulls the robot towards its goal configuration and a repellent potential field that pushes the robot away from obstacles.
No planning was involved in generating the potential field
The potential field can be calculated efficiently for any given configuration. Moreover, optimizing the potential amounts to calculating the gradient of the potential for the present robot configuration. These calculations can be extremely efficient, especially when compared to path-planning algorithms, all of which are exponential in the dimensionality of the configuration space (the DOFs) in the worst case.

Potential field control is great for local robot motion but sometimes we still need global planning.

Reactive control
control decisions that require some model of the environment for constructing either a reference path or a potential field.
First, models that are sufficiently accurate are often difficult to obtain, especially in complex or remote environments, such as the surface of Mars, or for robots that have few sensors. Second, even in cases where we can devise a model with sufficient accuracy, computational difficulties and localization error might render these techniques impractical.

A reflex agent architecture using **reactive control** is more appropriate
It is possible, nonetheless, to specify a controller directly without an explicit environmental model.

Variants of this simple feedback-driven controller have been found to generate remarkably robust walking patterns, capable of maneuvering the robot over rugged terrain

Reinforcement learning control
One particularly exciting form of control is based on the **policy search** form of reinforcement Learning. Policy search needs an accurate model of the domain before it can find a policy.

The controls are the manual controls of of the helicopter: throttle, pitch, elevator, aileron, and rudder. All that remains is the resulting state—how are we going to define a model that accurately says how the helicopter responds to each control? The answer is simple: Let an expert human pilot fly the helicopter, and record the controls that the expert transmits over the radio and the state variables of the helicopter. About four minutes of human-controlled flight suffices to build a predictive model that is sufficiently accurate to simulate the vehicle.